
164
SERIA
PC
DANIEL DANCIU
GEORGE MARDALE
ARTA
PROGRAMARII
ÎN JAVA
ALGORITMI ŞI
STRUCTURI
DE DATE
volumul II

Daniel Danciu **George Mardale**

ARTA PROGRAMĂRII ÎN JAVA

vol II - Algoritmi şi structuri de date

Cuprins

Introducere

Nu calculaţi capacitatea unui pod
numărând persoanele care
traversează acum râul înot.

Auzită la o prezentare

Oricine a folosit cel puţin o dată Internetul sau a citit o revistă de specialitate în domeniul informaticii, a auzit cu siguranţă cuvântul „Java". Java reprezintă un limbaj de programare, creat de compania Sun Microsystems în anul 1995. Iniţial, Java a fost gândit pentru a îmbunătăţi conţinutul paginilor web prin adăugarea unui conţinut dinamic: animaţie, multimedia etc. În momentul lansării sale, Java a revoluţionat Internetul, deoarece era prima tehnologie care oferea un astfel de conţinut. Ulterior au apărut şi alte tehnologii asemănătoare (cum ar fi Microsoft ActiveX sau Macromedia Shockwave[1]), dar Java şi-a păstrat importanţa deosebită pe care a dobândit-o în rândul programatorilor, în primul rând datorită facilităţilor pe care le oferea. Începând cu anul 1998, când a apărut versiunea 2 a limbajului (engl. Java 2 Platform), Java a fost extins, acoperind şi alte direcţii de dezvoltare: programarea aplicaţiilor *enterprise* (aplicaţii de tip server), precum şi a celor adresate dispozitivelor cu resurse limitate, cum ar fi telefoane mobile, *pager*-e sau PDA-uri[2]. Toate acestea au reprezentat facilităţi noi adăugate limbajului, care a păstrat însă şi posibilităţile de a crea aplicaţii standard, de tipul aplicaţiilor în linie de comandă sau aplicaţii bazate pe GUI[3].

[1]Cei care utilizează mai des Internetul sunt probabil obişnuiţi cu controale ActiveX sau cu animaţii flash în cadrul paginilor web.

[2]PDA = Personal Digital Assistant (mici dispozitive de calcul, de dimensiuni puţin mai mari decât ale unui telefon mobil, capabile să ofere facilităţi de agendă, dar şi să ruleze aplicaţii într-un mod relativ asemănător cu cel al unui PC). La momentul actual există mai multe tipuri de PDA-uri: palm-uri, pocketPC-uri, etc.

[3]GUI = Graphical User Interface, interfaţă grafică de comunicare cu utilizatorul, cum sunt în general aplicaţiile disponibile pe sistemul de operare Microsoft Windows.

Lansarea versiunii 2 a limbajului Java a fost o dovadă a succesului imens de care s-au bucurat versiunile anterioare ale limbajului, dar şi a dezvoltării limbajului în sine, a evoluţiei sale ascendente din punct de vedere al facilităţilor pe care le oferă, cât şi al performanţelor pe care le realizează.

Cum este organizată această carte?

Având în vedere popularitatea extraordinară de care se bucură limbajul Java în cadrul programatorilor din întreaga lume, am considerat utilă scrierea unei lucrări în limba română care să fie accesibilă celor care doresc să înveţe sau să aprofundeze acest limbaj. Ideea care a stat la baza realizării acestei cărţi a fost aceea de a prezenta nu numai limbajul Java în sine, ci şi modul în care se implementează algoritmii şi structurile de date fundamentale în Java, elemente care sunt indispensabile oricărui programator cu pretenţii. Aşadar, cartea nu este destinată numai celor care doresc să acumuleze noţiuni despre limbajul Java în sine, ci şi celor care intenţionează să îşi aprofundeze şi rafineze cunoştinţele despre algoritmi şi să îşi dezvolte un stil de programare elegant. Ca o consecinţă, am structurat cartea în două volume: prima volum (disponibil separat) este orientat spre prezentarea principalelor caracteristici ale limbajului Java, în timp ce volumul al doilea (cel de faţă) constituie o abordare a algoritmilor din perspectiva limbajului Java. Finalul primului volum cuprinde un grup de cinci anexe, care prezintă mai amănunţit anumite informaţii cu caracter mai special, deosebit de utile pentru cei care ajung să programeze în Java. Am ales această strategie deoarece a dobândi cunoştinţe despre limbajul Java, fără a avea o imagine clară despre algoritmi, nu reprezintă un progres real pentru un programator. Iar scopul nostru este acela de a vă oferi posibilitatea să deveniţi un programator pentru care limbajul Java şi algoritmii să nu mai constituie o necunoscută.

Cele două volume cuprind numeroase soluţii Java complete ale problemelor prezentate. Mai este necesară o remarcă: deseori am optat, atât în redactarea codului sursă cât şi în prezentarea teoretică a limbajului sau a algoritmilor, pentru păstrarea terminologiei în limba engleză în defavoarea limbii române. Am luat această decizie, ţinând cont de faptul că mulţi termeni s-au consacrat în acest format, iar o eventuală traducere a lor le-ar fi denaturat înţelesul.

Primul volum al cărţii cuprinde opt capitole:

Capitolul 1 reprezintă o prezentare de ansamblu a tehnologiei Java. Capitolul debutează cu istoria limbajului, începând cu prima versiune şi până la cea curentă. În continuare, sunt înfăţişate câteva detalii despre implementările existente ale limbajului. Implementarea Java a firmei Sun este tratată separat, în detaliu, fiind şi cea pe care s-au testat aplicaţiile realizate pe parcursul cărţii.

Capitolul 2 este cel care dă startul prezentării propriu-zise a limbajului Java,

începând cu crearea şi executarea unui program simplu. Sunt prezentate apoi tipurile primitive de date, constantele, declararea şi iniţializarea variabilelor, operatorii de bază, conversiile, instrucţiunile standard şi metodele.

Capitolul 3 este destinat referinţelor şi obiectelor. Sunt prezentate în detaliu noţiunile de referinţă, obiect, şiruri de caractere (String-uri) şi de şiruri de elemente cu dimensiuni variabile.

Capitolul 4 continuă prezentarea începută în capitolul anterior, prezentând modul în care se pot defini propriile clase în Java şi cum se implementează conceptele fundamentale ale programării orientate pe obiecte.

Capitolul 5 prezintă în detaliu un principiu esenţial al programării orientate pe obiecte: *moştenirea*. Sunt prezentate de asemenea noţiuni adiacente cum ar fi cea de polimorfism, interfaţă, clasă interioară, identificare a tipurilor de date în faza de execuţie (RTTI = Runtime Type Identification).

Capitolul 6 este dedicat în întregime modului de tratare a excepţiilor în Java. Se prezintă tipurile de excepţii existente în limbajul Java, cum se pot defini propriile tipuri de excepţii, cum se pot prinde şi trata excepţiile aruncate de o aplicaţie. Finalul capitolului este rezervat unei scurte liste de sugestii referitoare la utilizarea eficientă a excepţiilor.

Capitolul 7 prezintă sistemul de intrare-ieşire (I/O) oferit de limbajul Java. Pe lângă operaţiile standard realizate pe fluxurile de date (*stream*) şi fişiere secvenţiale, este prezentată şi noţiunea de *colecţie de resurse* (engl. resource bundles).

Capitolul 8 este rezervat problemei delicate a firelor de execuţie (*thread*-uri). Pornind cu informaţii simple despre firele de execuţie, se continuă cu accesul concurent la resurse, sincronizare, monitoare, coordonarea firelor de execuţie, cititorul dobândind în final o imagine completă asupra sistemului de lucru pe mai multe fire de execuţie, aşa cum este el atât de elegant realizat în Java.

Cele opt capitole ale primului volum sunt urmate de un grup de anexe, care conţin multe informaţii utile programatorilor Java.

Anexa A constituie o listă cu editoarele şi mediile integrate pentru dezvoltarea aplicaţiilor Java, precum şi un mic tutorial de realizare şi executare a unei aplicaţii Java simple. Tot aici este prezentată şi *ant*, o unealtă de foarte mare ajutor în compilarea şi rularea aplicaţiilor de dimensiuni mai mari.

Anexa B este dedicată convenţiilor de scriere a programelor. Sunt prezentate principalele reguli de scriere a unor programe lizibile, conforme cu standardul stabilit de Sun Microsystems. Ultima parte a anexei este dedicată unei unelte foarte utile în documentarea aplicaţiilor Java: *javadoc*.

Anexa C detaliază ideea de pachete, oferind informaţii despre pachete Java

predefinite şi despre posibilitatea programatorului de a defini propriile sale pachete. Un accent deosebit se pune şi pe prezentarea arhivelor `jar`.

Anexa D prezintă modul în care se pot realiza aplicaţii internaţionalizate, prin care textele care apar într-o aplicaţie sunt traduse dintr-o limbă în alta, cu un efort minim de implementare. De asemenea, este prezentat şi rolul colecţiilor de resurse (engl. resource bundles) în internaţionalizarea aplicaţiilor.

Anexa E reprezintă o listă de resurse disponibile programatorului Java, pornind de la *site*-ul Sun Microsystems şi până la tutoriale, cărţi, reviste online, liste de discuţii disponibile pe Internet. Cu ajutorul acestora, un programator Java poate să îşi dezvolte aptitudinile de programare şi să acumuleze mai multe cunoştinţe despre limbajul Java.

Volumul de faţă, al doilea al cărţii, este destinat prezentării algoritmilor. Independenţa algoritmilor relativ la un anumit limbaj de programare, face ca majoritatea programelor din această parte să fie realizate şi în pseudocod, punctând totuşi pentru fiecare în parte specificul implementării în Java.

Capitolul 9 constituie primul capitol al celui de-al doilea volum şi prezintă modalitatea prin care se poate realiza analiza eficienţei unui algoritm. Notaţia asimptotică, tehnicile de analiză a algoritmilor, algoritmii recursivi constituie principala direcţie pe care se axează acest capitol.

Capitolul 10 reprezintă o incursiune în cadrul structurilor de date utilizate cel mai frecvent în conceperea algoritmilor: stive, cozi, liste înlănţuite, arbori binari de căutare, tabele de repartizare şi cozi de prioritate. Fiecare dintre aceste structuri beneficiază de o prezentare în detaliu, însoţită de o implementare Java în care se pune accent pe separarea interfeţei structurii de date de implementarea acesteia.

Capitolul 11 constituie startul unei suite de capitole dedicate metodelor fundamentale de elaborare a algoritmilor. Primul capitol din această suită este rezervat celei mai elementare metode: *backtracking*. După o analiză amănunţită a caracteristicilor acestei metode (cum ar fi cei patru paşi standard în implementarea metodei: *atribuie şi avansează*, *încercare eşuată*, *revenire*, *revenire după construirea unei soluţii*), sunt prezentate câteva exemple de probleme clasice care admit rezolvare prin metoda backtracking: generarea permutărilor, a aranjamentelor şi a combinărilor, problema damelor, problema colorării hărţilor.

Capitolul 12 prezintă o altă metodă de elaborare a algoritmilor: *divide et impera*. Prima parte a capitolului este rezervată prezentării unor noţiuni introductive despre recursivitate şi recurenţă, absolut necesare înţelegerii modului în care funcţionează această metodă. Analog capitolului 11, prezentarea propriuzisă a metodei este însoţită de câteva exemple de probleme clasice rezolvabile prin această metodă: căutarea binară, sortarea prin interclasare (*mergesort*),

sortarea rapidă (*quicksort*), trecerea expresiilor aritmetice în formă poloneză postfixată.

Capitolul 13 prezintă cea de-a treia metodă de elaborare a algoritmilor: *metoda Greedy*. Capitolul păstrează aceeaşi structură ca şi cele precedente: sunt prezentate mai întâi elementele introductive ale metodei, urmate apoi de câteva exemple clasice de probleme rezolvabile prin această metodă: problema spectacolelor, minimizarea timpului de aşteptare, interclasarea optimă a mai multor şiruri ordonate.

Capitolul 14 este rezervat unei metode speciale de elaborare a algoritmilor: *programarea dinamică*, ce reprezintă probabil cea mai complexă metodă de elaborare a algoritmilor, punând deseori în dificultate şi programatorii experimentaţi. Totuşi avem credinţa că modul simplu şi clar în care sunt prezentate noţiunile să spulbere mitul care înconjoară această metodă. Capitolul debutează cu o fundamentare teoretică a principalelor concepte întâlnite în cadrul metodei. Apoi, se continuă cu rezolvarea unor probleme de programare dinamică: înmulţirea unui şir de matrice, subşirul comun de lungime maximă, distanţa Levensthein etc.

Capitolul 15 reprezintă ultimul capitol din seria celor dedicate metodelor de elaborare a algoritmilor. Metoda *branch and bound* este cea abordată în cadrul acestui capitol, prin intermediul unui exemplu clasic de problemă: jocul de *puzzle* cu 15 elemente.

Capitolul 16 reprezintă o sinteză a metodelor de elaborare a algoritmilor care au fost tratate de-a lungul volumului al doilea, prezentând aspecte comune şi diferenţe între metode, precum şi aria de aplicabilitate a fiecărei metode în parte.

Esenţa acestui volum o reprezintă metodele fundamentale de elaborare a algoritmilor împreună cu structurile de date cele mai uzuale, precum şi modul în care acestea se particularizează pentru a fi implementate în limbajul Java.

Primul capitol al acestei părţi introduce noţiuni esenţiale referitoare la analiza eficienţei algoritmilor, noţiuni care vor fi ulterior folosite pentru a evalua eficienţa diferitelor operaţii pe structuri de date, precum şi a algoritmilor prezentaţi. Următorul capitolul introduce succesiv structurile de date cele mai uzuale, începând cu listele şi încheind cu structuri ceva mai complexe cum ar fi arborii sau cozile de prioritate.

Restul capitolelor (de la capitolul 11 până la capitolul 15) prezintă pe rând metodele fundamentale de elaborare a algoritimilor care reprezintă nişte tehnici cu caracter general prin care se poate rezolva o anumită clasă largă de probleme. Aceste metode nu sunt legate de un limbaj de programare anume, şi din acest motiv mulţi autori preferă să le trateze la modul general, descriind re-

zolvarea problemelor în pseudocod. În această lucrare am urmărit să prezentăm metodele de elaborare a algoritmilor la modul general, precum şi specificul dat de implementarea rezolvărilor în limbajul Java.

Metodele de elaborare a algoritmilor prezentate în această parte sunt:

- **Backtracking***:* se aplică problemelor a căror soluţie se poate scrie sub formă de vector. Această metodă construieşte vectorul soluţie componentă cu componentă, cu eventuale reveniri asupra componentelor anterioare;

- **Greedy**: principiul acestei metode este asemănător cu cel de la *backtracking*, cu diferenţa că selectarea următorului pas se face pe baza unui criteriu local fără a se reveni asupra paşilor anteriori;

- **Divide et impera** (dezbină şi cucereşte): această metodă împarte problema originală în două sau mai multe subprobleme; subproblemele sunt împărţite la rândul lor în sub-subprobleme şi aşa mai departe până când se ajunge la subprobleme de dimensiune mică, a căror rezolvare este trivială. Se construieşte apoi soluţia problemei originale prin combinarea soluţiilor subproblemelor. Evident, nu orice problemă poate fi rezolvată în acest mod;

- **Programare dinamică**: aplicabilă doar problemelor de optimizare (sau care pot fi reformulate ca probleme de optimizare) care respectă aşa-numitul *principiu al optimalităţii*;

- **Branch & bound**: termen care ar putea fi tradus cu aproximaţie prin "împarte şi evaluează". Este o variantă a tehnicii *backtracking*, în care alegerea următorului pas nu se face la întâmplare, ci într-o anumită ordine dată de o evaluare locală a şanselor ca acel pas să conducă la o soluţie.

Metodele de elaborare a algoritmilor au fost concepute ca nişte tehnici cu caracter general aplicabile unei clase foarte largi de probleme de programare. Majoritatea problemelor de programare pot fi abordate cu una sau mai multe din aceste metode de elaborare a algoritmilor şi, astfel, programatorul nu este întotdeauna nevoit să conceapă câte o metodă ad-hoc pentru fiecare problemă pe care o are de rezolvat. Programatorul trebuie să încadreze problema pe care o are de rezolvat în una dintre aceste metode de elaborare, după care particularizează acea metodă pentru problema concretă şi alege structurile de date adecvate.

În linii mari, putem spune că rezolvarea unei probleme de programare presupune următorii paşi:

1. Identificarea metodei de elaborare şi a structurilor de date potrivite;

2. Particularizarea acestei tehnici pentru problema concretă.

În acest punct, tehnicile de programare se diferenţiază: pentru unele metode (cum ar fi *backtracking*) trecerea de la forma generală la forma concretă este aproape algoritmică, nefiind necesar un mare efort de adaptare, în timp ce pentru altele (cum ar fi *programarea dinamică*) trecerea necesită un efort considerabil, dublat de ingeniozitate şi o profundă stăpânire a metodei.

Cui se adresează această carte?

Lucrarea de faţă nu se adresează începătorilor, ci persoanelor care stăpânesc deja, chiar şi parţial, un limbaj de programare. Cititorii care au cunoştinţe de programare în C şi o minimă experienţă de programare orientată pe obiecte vor găsi lucrarea ca fiind foarte uşor de parcurs. Nu sunt prezentate noţiuni elementare specifice limbajelor de programare cum ar fi funcţii, parametri, instrucţiuni etc. Nu se presupune cunoaşterea unor elemente legate de programarea orientată pe obiecte, deşi existenţa lor poate facilita înţelegerea noţiunilor prezentate. De asemenea, cartea este foarte utilă şi celor care doresc să aprofundeze studiul algoritmilor şi modul în care anumite probleme clasice de programare pot fi implementate în Java.

Pe Internet

Pentru comoditatea cititorilor, am decis să punem la dispoziţia lor codul sursă complet al tuturor programelor prezentate pe parcursul celor două volume ale lucrării în cadrul unei pagini web concepută special pentru interacţiunea cu cititorii. De asemenea, pagina web a cărţii va găzdui un forum unde cititorii vor putea oferi sugestii în vederea îmbunătăţirii conţinutului lucrării, vor putea schimba opinii în legătură cu diversele aspecte prezentate, adresa întrebări autorilor etc. Adresele la care veţi găsi aceste informaţii sunt:

- `http://www.albastra.ro/carti/v178/`
- `http://www.danciu.ro/apj/`

Mulţumiri

În încheiere, dorim să adresăm mulţumiri colegilor şi prietenilor noştri care ne-au acordat ajutorul în realizarea acestei lucrări: Vlad Petric (care a avut o

contribuţie esenţială la structurarea capitolului 14), Alexandru Băluţ (autor al anexei A), Iulia Tatomir (a parcurs şi comentat cu multă răbdare de mai multe ori întreaga carte) şi Raul Furnică (a parcurs şi comentat capitolele mai delicate ale lucrării).

9. Analiza eficienţei algoritmilor

> Nu vă faceţi griji pentru problemele pe care vi le pune matematica. Vă asigur că ale mele sunt cu mult mai mari.
>
> Albert Einstein

În prima parte a cărţii am examinat cum putem folosi programarea orientată pe obiecte pentru proiectarea şi implementarea programelor profesionale. Au fost prezentate trăsăturile fundamentale ale limbajului Java, precum şi facilităţi mai puţin cunoscute, dar foarte utile, cum ar fi mecansimul de reflectare (Reflection API) sau colecţiile de resurse. Totuşi, aceasta reprezintă doar jumătate din problemă.

Calculatorul este folosit de obicei pentru a prelucra cantităţi mari de informaţie. Atunci când executăm un program cu date de intrare de dimensiuni mari, trebuie să fim siguri că vom obţine rezultatul într-un timp rezonabil. Acest lucru este aproape întotdeauna independent de limbajul de programare folosit, ba chiar şi de metodologia aplicată (cum ar fi programare orientată pe obiecte, sau programare procedurală).

Un algoritm este un set bine precizat de instrucţiuni pe care calculatorul le va executa pentru a rezolva o problemă. Odată ce am găsit un algoritm pentru o anumită problemă şi am determinat că algoritmul este corect, pasul următor este de a determina cantitatea de resurse, cum ar fi timpul şi cantitatea de memorie, pe care algoritmul le cere. Acest pas este numit analiza algoritmului. Un algoritm care are nevoie de câţiva *gigabytes* de memorie nu este bun de nimic pe calculatoarele existente la ora actuală, chiar dacă el este corect.

În acest capitol vom vedea:

- Cum putem estima timpul cerut de un algoritm (altfel spus, determinarea complexităţii algoritmului);

- Tehnici pentru reducerea drastică a timpului de execuţie al unui algoritm;
- Un cadru matematic care descrie la un mod mai riguros timpul de execuţie al algoritmilor.

9.1 Ce este analiza algoritmilor?

Cantitatea de timp pe care orice algoritm o cere pentru execuţie depinde aproape întotdeauna de cantitatea de date de intrare pe care o procesează. Este de aşteptat că sortarea a 10.000 de elemente să necesite mai mult timp decât sortarea a 10 elemente. Timpul de execuţie al unui algoritm este astfel o funcţie de dimensiunea datelor de intrare. Valoarea exactă a acestei funcţii depinde de mai mulţi factori, cum ar fi viteza calculatorului pe care rulează programul, calitatea compilatorului şi, nu de puţine ori, calitatea programului. Pentru un program dat, care rulează pe un anumit calculator, putem reprezenta grafic timpul de execuţie. În **Figura 9.1** am realizat un astfel de grafic pentru patru programe. Curbele reprezintă patru funcţii care sunt foarte des întâlnite în analiza algoritmilor: *liniară*, $n \log n$, *pătratică* şi *cubică*. Dimensiunea datelor de intrare variază de la 1 la 100 de elemente, iar timpii de execuţie de la 0 la 5 milisecunde. O privire rapidă asupra graficelor din **Figura 9.1** şi **Figura 9.2** ne lămureşte că ordinea preferinţelor pentru timpii de execuţie este liniar, $n \log n$, pătratic şi cubic.

Să luăm ca exemplu descărcarea (download-area) unui fişier de pe Internet. Să presupunem că la început apare o întârziere de două secunde (pentru a stabili conexiunea), după care descărcarea se va face la 1.6 KB/sec. În această situaţie, dacă fişierul de adus are `N` kilobaiţi, timpul de descărcare a fişierului este descris de formula `T(N) = N/1.6 + 2`. Aceasta este o funcţie *liniară*. Se poate calcula uşor că descărcarea unui fişier de 80K va dura aproximativ 52 de secunde, în timp ce descărcarea unui fişier de două ori mai mare (160K) va dura 102 secunde, deci cam de două ori mai mult. Această proprietate, în care timpul este practic direct proporţional cu cantitatea de date de intrare, este specifică unui *algoritm liniar*, şi constituie adeseori o situaţie ideală. Aşa cum se vede din grafice, unele curbe neliniare pot conduce la timpi de execuţie foarte mari.

În acest capitol vom analiza următoarele probleme: cu cât este mai bună o curbă în comparaţie cu o altă curbă, cum putem calcula pe care curbă se situează un anumit algoritm sau cum putem proiecta algoritmi care să nu se situeze pe curbele nefavorabile.

O funcţie *cubică* este o funcţie al cărei termen dominant este N^3, înmulţit cu o constantă. De exemplu, $10N^3 + N^2 + 40N + 80$ este o funcţie cubică.

Figura 9.1: Timpi de execuţie pentru date de dimensiune mică

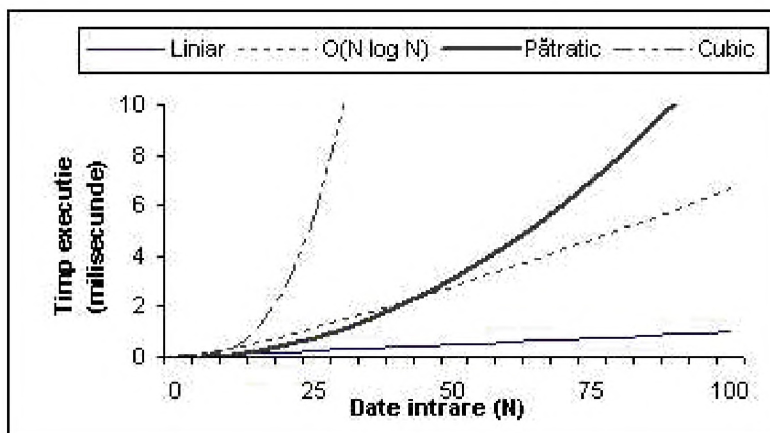
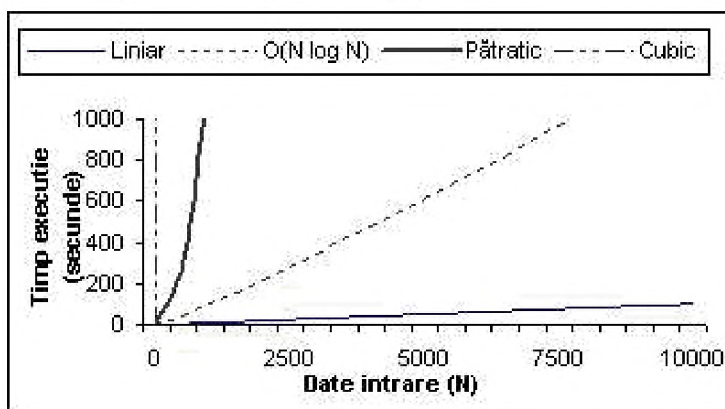


Figura 9.2: Timpi de execuţie pentru date de intrare moderate



Similar, o funcţie pătratică are termenul dominant N^2 înmulţit cu o constantă, iar o funcţie liniară are un termen dominant care este N înmulţit cu o constantă.

Oricare dintre cele trei funcţii prezentate mai sus poate fi mai mică decât cealaltă într-un punct dat. Acesta este motivul pentru care nu ne interesează valorile efective ale timpilor de execuţie, ci rata lor de creştere. Acest lucru este justificabil prin trei argumente. În primul rând, pentru funcţiile cubice, cum ar fi cea prezentată în **Figura 9.2**, atunci când N are valoarea 1000, valoarea funcţiei cubice este aproape complet determinată de valoarea termenului cubic. Funcţia $10N^3+N^2+40N+80$ are valoarea 10.001.040.080 pentru $N = 1000$, din care 10.000.000.000 se datorează termenului $10N^3$. Dacă am fi folosit doar termenul cubic pentru a estima valoarea funcţiei, ar fi rezultat o eroare de aproximativ 0.01%. Pentru un N suficient de mare, valoarea funcţiei este determinată aproape complet de termenul ei dominant (semnificaţia termenului *suficient de mare* depinde de funcţia în cauză).

Un al doilea motiv pentru care măsurăm doar rata de creştere a funcţiilor este că valoarea exactă a constantei multiplicative pentru termenul dominant diferă de la un calculator la altul. De exemplu, calitatea compilatorului poate să influenţeze destul de mult valoarea constantei. În al treilea rând, valorile mici pentru N sunt de obicei nesemnificative. Din **Figura 9.1** se observă că pentru $N = 10$, toţi algoritmii se încheie în mai puţin de 3 ms. Diferenţa dintre cel mai bun şi cel mai slab algoritm este mai mică decât un clipit de ochi.

Pentru a reprezenta rata de creştere a unui algoritm se foloseşte aşa-numita notaţie asimptotică (engl. "Big-Oh notation"). De exemplu, rata de creştere pentru un algoritm pătratic este notată cu O(N^2). Notaţia asimptotică ne permite să stabilim o ordine parţială între funcţii prin compararea termenului lor dominant.

Vom dezvolta în acest capitol aparatul matematic necesar pentru analiza eficienţei algoritmilor, urmărind ca această incursiune matematică să nu fie excesiv de formală. Vom arăta apoi, pe bază de exemple, cum poate fi analizat un algoritm. O atenţie specială o vom acorda tehnicilor de analiză a algoritmilor recursivi.

9.2 Notaţia asimptotică

Notaţia asimptotică are rolul de a estima timpul de calcul necesar unui algoritm pentru a furniza rezultatul, funcţie de dimensiunea datelor de intrare.

9.2.1 O notaţie pentru ordinul de mărime al timpului de execuţie al unui algoritm

Fie **N** mulţimea numerelor naturale, $\Re$ mulţimea numerelor reale. Fie f : $\mathbf{N} \rightarrow [0, \infty)$ o funcţie arbitrară. Definim mulţimea de funcţii:

$$O(f) = \{t : \mathbf{N} \rightarrow [0, \infty) \mid \exists c > 0, \exists n_0 \in \mathbf{N}, \quad astfel \quad incat \quad \forall \quad n \geq n_0 \quad avem \quad t(n) \leq c * f(n)\}$$

Cu alte cuvinte, *O(f)* (se citeşte "*ordinul lui f*") este mulţimea tuturor funcţiilor *t* mărginite superior de un multiplu real pozitiv al lui *f*, pentru valori suficient de mari ale argumentului *n*. Vom conveni să spunem că *t* este în ordinul lui *f* (sau, echivalent, *t* este în *O(f)*, sau $t \in O(f)$) chiar şi atunci când *t(n)* este negativ sau nedefinit pentru anumite valori $n < n_0$. În mod similar, vom vorbi despre ordinul lui *f* chiar şi atunci când valoarea *f(n)* este negativă sau nedefinită pentru un număr finit de valori ale lui *n*; în acest caz, vom alege n_0 suficient de mare, astfel încât pentru $n \geq n_0$ acest lucru să nu mai apară. De exemplu, vom vorbi despre ordinul lui *n/log n* , chiar dacă pentru *n=0* şi *n=1* funcţia nu este definită. În loc de $t \in O(f)$, uneori este mai convenabil să folosim notaţia $t(n) \in O(f(n))$, subînţelegând aici că *t(n)* şi *f(n)* sunt funcţii.

Fie un algoritm dat şi fie o funcţie $t : \mathbf{N} \rightarrow [0, \infty)$, astfel încât o anumită implementare a algoritmului să necesite cel mult *t(n)* unităţi de timp pentru a rezolva un caz de mărime n, unde $n \in \mathbf{N}$. Principiul invarianţei[1] ne asigură atunci că orice implementare a algoritmului necesită un timp în ordinul lui *t*. Cu alte cuvinte, acest algoritm necesită un timp în ordinul lui *f* pentru orice funcţie $f : \mathbf{N} \rightarrow [0, \infty)$ pentru care $t \in O(f)$. În particular avem relaţia: $t \in O(t)$. Vom căuta, în general, să găsim cea mai simplă funcţie *f*, astfel încât $t \in O(f)$.

Exemplu: Fie funcţia $t(n) = 3n^2 - 9n + 13$. Pentru n suficient de mare, vom avea relaţia $t(n) \leq 4n^2$. În consecinţă, luând $c = 4$, putem spune că $t(n) \in O(n^2)$. La fel de bine puteam să spunem că $t(n) \in O(13n^2 - \sqrt{2}n + 12.5)$, dar pe noi ne interesează să găsim o expresie cât mai simplă. Este adevărată şi relaţia $t(n) \in O(n^4)$ dar, aşa cum vom vedea mai târziu, suntem interesaţi de a mărgini cât mai strâns ordinul de mărime al algoritmului, pentru a putea obiectiva cât mai bine durata sa de execuţie.

Proprietăţile de bază ale lui *O(f)* sunt date ca exerciţii (1 - 5) şi ar fi recomandabil să le studiaţi înainte de a trece mai departe.

[1] Acest principiu afirmă că două implementări diferite ale aceluiaşi algoritm nu diferă, ca eficienţă, decât cel mult printr-o constantă multiplicativă.

Notaţia asimptotică defineşte o relaţie de ordine parţială între funcţii şi, prin urmare, între eficienţa relativă a diferiţilor algoritmi care rezolvă o anumită problemă. Vom da în continuare o interpretare algebrică a notaţiei asimptotice. Pentru oricare două funcţii $f, g : \mathbf{N} \rightarrow \Re^*$ definim următoarea relaţie binară: $f \leq g$ dacă $O(f) \subseteq O(g)$. Relaţia "$\leq$ " este o *relaţie de ordine parţială (reflexivă, tranzitivă, antisimetrică)* în mulţimea funcţiilor definite pe $\mathbf{N}$ şi cu valori în $[0, \infty)$ (exerciţiul 4). Definim şi o *relaţie de echivalenţă*: $f \equiv g$ dacă $O(f) = O(g)$. Prin această relaţie obţinem clase de echivalenţă, o *clasă de echivalenţă* cuprinzând toate funcţiile care diferă între ele printr-o constantă multiplicativă. De exemplu, $lg\, n \equiv ln\, n$ şi avem o clasă de echivalenţă a funcţiilor logaritmice, pe care o notăm generic cu $O(log\ n)$. Notând cu $O(1)$ clasa de echivalenţă a algoritmilor cu timpul mărginit superior de o constantă (cum ar fi interschimbarea a două numere, sau maximul a trei elemente), ierarhia celor mai cunoscute clase de echivalenţă este:

$$O(1) \subset O(log\, n) \subset O(n) \subset O(nlog\, n) \subset O(n^2) \subset O(n^3) \subset O(2^n)$$

Această ierarhie corespunde unei clasificări a algoritmilor după un criteriu al performanţei. Pentru o problemă dată, dorim mereu să obţinem un algoritm corespunzător unei clase aflate cât mai "de jos" (cu timp de execuţie cât mai mic). Astfel, se consideră a fi o mare realizare dacă în locul unui algoritm exponenţial găsim un algoritm polinomial.

Exerciţiul 5 ne dă o metodă de simplificare a calculelor în care apare notaţia asimptotică. De exemplu:

$$\begin{gathered} n^3 + 4n^2 + 2n + 7 \in O(n^3 + (4n^2 + 2n + 7)) = \\ O(max(n^3, 4n^2 + 2n + 7)) = O(n^3) \end{gathered}$$

Ultima egalitate este adevărată chiar dacă $max(n^3, 4n^2 + 2n + 7) \neq n^3$ pentru $0 \leq n \leq 4$, deoarece *notaţia asimptotică* se aplică doar pentru *n* suficient de mare. De asemenea,

$$\begin{gathered} n^3 - 3n^2 - n - 8 \in O(\tfrac{n^3}{2} + (\tfrac{n^3}{2} - 3n^2 - n - 8)) = \\ O(max(\tfrac{n^3}{2}, \tfrac{n^3}{2} - 3n^2 - n - 8)) = O(\tfrac{n^3}{2}) = O(n^3) \end{gathered}$$

chiar dacă pentru $0 \leq n \leq 6$ polinomul este negativ. Exerciţiul 6 tratează cazul unui polinom oarecare.

Notaţia $O(f)$ este folosită pentru a limita superior timpul necesar unui algoritm, măsurând eficienţa (complexitatea computaţională) a algoritmului respectiv. Uneori este interesant să estimăm şi o limită inferioară a acestui timp. În acest scop, definim mulţimea:

$$\Omega(f) = \{t : \mathbf{N} \to [0,\infty) \mid \exists c > 0, \exists n_0 \in \mathbf{N}, \quad astfel \quad incat \quad \forall \quad n \geq n_0 \quad avem \quad t(n) \geq c * f(n)\}$$

Există o anumită dualitate între notaţiile *O(f)* şi $\Omega(f)$: pentru două funcţii oarecare $f, g : \mathbf{N} \to [0,\infty)$, avem:

$$f \in O(g) \text{ dacă şi numai dacă } g \in \Omega(f) .$$

O estimare foarte precisă a timpului de execuţie se obţine atunci când timpul de execuţie al unui algoritm este limitat atât inferior cât şi superior de câte un multiplu real pozitiv al aceleaşi funcţii. În acest scop, introducem notaţia:

$$\Theta(f) = O(f) \cap \Omega(f)$$

numită *ordinul exact al lui f*. Pentru a compara ordinele a două funcţii, notaţia Θ nu este însă mai puternică decât notaţia *O*, în sensul că *O(f)=O(g)* este echivalent cu $\Theta(f) = \Theta(g)$.

Există situaţii în care timpul de execuţie al unui algoritm depinde simultan de mai mulţi parametri. Aceste situaţii sunt tipice pentru anumiţi algoritmi care operează cu grafuri şi la care timpul depinde atât de numărul de vârfuri cât şi de numărul de muchii. Notaţia asimptotică se generalizează în mod natural şi pentru funcţii cu mai multe variabile. Astfel, pentru o funcţie arbitrară $f : \mathbf{N} \times \mathbf{N} \to [0,\infty)$ definim

$$O(f) = \{t : \mathbf{N} \times \mathbf{N} \to [0,\infty) \mid \exists c > 0, \exists n_0, m_0 \in \mathbf{N}, \quad astfel \quad incat \quad \forall \quad m \geq m_0, \forall \quad n \geq n_0 \quad avem \quad t(m,n) \leq c * f(m,n)\}.$$

Similar se obţin şi celelalte generalizări.

9.3 Tehnici de analiza algoritmilor

Nu există o metodă standard pentru analiza eficienţei unui algoritm. Este mai curând o chestiune de raţionament, intuiţie şi experienţă. Vom arăta pe bază de exemple cum se poate efectua o astfel de analiză.

9.3.1 Sortarea prin selecţie

Considerăm algoritmul de sortare prin selecţia minimului, reprodus mai jos:

<u>*pentru*</u> $i = 1, n-1$
    //se calculează poziţia minimului lui $a(i), a(i+1), \ldots, a(n)$
    $PozMin \leftarrow i$ *//initializăm minimul cu indicele primului element*
    <u>*pentru*</u> $j = i+1, n$
        <u>*dacă*</u> $a(j) < a(PozMin)$ <u>*atunci*</u>
            $PozMin = j$
        <u>*sfârşit dacă*</u>
    <u>*sfârşit pentru*</u> *//după j*
    //se aşează minimul pe poziţia i
    $aux \leftarrow a(i)$
    $a(i) \leftarrow a(PozMin)$
    $a(PozMin) \leftarrow aux$
<u>*sfârşit pentru*</u> *//după i*

Timpul necesar pentru o singură execuţie a ciclului *pentru* după variabila *j* poate fi mărginit superior de o constantă a. În total, pentru un i fixat, ţinând cont de faptul că se realizează *n-i* iteraţii, acest ciclu necesită un timp de cel mult $b + a(n-i)$ unităţi, unde *b* este o constantă reprezentând timpul necesar pentru iniţializarea buclei. O singură execuţie a buclei exterioare are loc în cel mult $c + b + a(n-i)$ unităţi de timp, unde *c* este o altă constantă. Ţinând cont de faptul că bucla după *j* se realizează de *n-1* ori, timpul total de execuţie al algoritmului este cel mult:

$$d + \sum_{i=1}^{n-1} (c + b + a(n-i))$$

unităţi de timp, d fiind din nou o constantă. Simplificăm această expresie şi obţinem $\frac{a}{2}n^2 + (b + c - \frac{a}{2})n + (d - c - b)$, de unde deducem că algoritmul necesită un timp în $O(n^2)$. O analiză similară asupra limitei inferioare arată că timpul este de fapt în $\Theta(n^2)$. Nu este necesar să considerăm cazul cel mai nefavorabil sau cazul mediu deoarece timpul de execuţie al sortării prin selecţie este independent de ordonarea prealabilă a elementelor de sortat.

În acest prim exemplu am analizat toate detaliile. De obicei însă, detalii cum ar fi timpul necesar iniţializarii ciclurilor nu se vor considera explicit, deoarece ele nu afectează ordinul de complexitate al algoritmului. Pentru cele mai multe situaţii, este suficient să alegem o anumită instrucţiune din algoritm ca *barometru* şi să numărăm de câte ori se execută această instrucţiune. În cazul nostru, putem alege ca barometru testul

$$a[j] < a[PozMin]$$

din bucla interioară. Este uşor de observat că acest test se execută de $\frac{n(n-1)}{2}$ ori.

9.3.2 Sortarea prin inserţie

Timpul pentru *algoritmul de sortare prin inserţie* este dependent de ordonarea prealabilă a elementelor de sortat. Algoritmul este implementat în cadrul primului volum, la capitolul *Moştenire*, secţiunea *Extinderea clasei Shape*. Analiza algoritmului se realizează pe baza implementării prezentate în cadrul acelei secţiuni. Vom folosi comparaţia

```
tmp.lessThan(a[j - 1])
```

din ciclul *for* ca barometru.

Să presupunem că p este fixat şi fie $n = a.length$ lungimea şirului. Cel mai nefavorabil caz apare atunci când $tmp < a[j-1]$ pentru fiecare j între p şi 1, algoritmul făcând în această situaţie p comparaţii. Acest lucru se întâmplă (pentru fiecare valoare a lui p de la 1 la $n-1$) atunci când tabloul a este iniţial ordonat descrescător. Numărul total de comparaţii pentru cazul cel mai nefavorabil este:

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \in \Theta(n^2)$$

Vom estima acum timpul mediu necesar pentru un caz oarecare. Presupunem că elementele tabloului a sunt distincte şi că orice permutare a lor are aceeaşi probabilitate de apariţie. Atunci, dacă $1 \leq k \leq p$, probabilitatea ca $a[p]$ să fie cel de-al k-lea cel mai mare element dintre elementele $a[1], a[2], \ldots, a[p]$ este $\frac{1}{p}$. Pentru un p fixat, condiţia $a[p] < a[p-1]$ este falsă cu probabilitatea $\frac{1}{p}$, deci probabilitatea ca să se execute comparaţia "$tmp < a[j-1]$" o singură dată înainte de ieşirea din bucla *while* este $\frac{1}{p}$. Comparaţia "$tmp < a[j-1]$" se execută de exact două ori tot cu probabilitatea $\frac{1}{p}$ etc. Probabilitatea ca să se execute comparaţia de exact $p-1$ ori este $\frac{2}{p}$, deoarece aceasta se întâmplă atât când $tmp < a[0]$ cât şi când $a[0] \leq tmp < a[1]$! Numărul mediu de comparaţii, pentru un p fixat, este în consecinţă, suma numărului de comparaţii pentru fiecare situaţie, înmulţită cu probabilitatea de apariţie a acelei situaţii:

$$c_i = 1\frac{1}{i} + 2\frac{1}{i} + \ldots + (i-2)\frac{1}{i} + (i-1)\frac{2}{i} = \frac{i+1}{2} - \frac{1}{i}$$

Pentru a sorta n elemente avem nevoie de $\sum_{i=2}^{n} c_i$ comparaţii, ceea ce este egal cu $\frac{n^2+3n}{4} - H_n \in \Theta(n^2)$. Prin $H_n = \sum_{i=1}^{n} i^{-1} \in \Theta(log\, n)$ am notat al n-lea termen al seriei armonice.

Se observă că algoritmul de sortare prin inserare efectuează pentru cazul mediu de două ori mai puţine comparaţii decât pentru cazul cel mai nefavorabil. Totuşi, în ambele situaţii, numărul comparaţiilor este în $\Theta(n^2)$.

Cu toate că algoritmul necesită un timp în $\Omega(n^2)$ atât pentru cazul mediu cât şi pentru cel mai nefavorabil caz, pentru cazul cel mai favorabil (când iniţial tabloul este ordonat crescător) timpul este în $O(n)$. De fapt, pentru cazul cel mai favorabil, timpul este şi în $\Omega(n)$ (deci în $\Theta(n)$).

9.3.3 Turnurile din Hanoi

Matematicianul francez Eduard Lucas a propus în 1883 o problemă care a devenit apoi celebră mai ales datorită faptului că a prezentat-o sub forma unei legende. Se spune că Brahma (Zeul Creaţiei la hinduşi) a fixat pe Pământ trei tije de diamant şi pe una din ele a pus în ordine crescătoare 64 de discuri de aur de dimensiuni diferite, astfel încât discul cel mai mare era jos. Brahma a creat şi o mănăstire, iar sarcina călugărilor era să mute toate discurile pe o altă tijă. Singura operaţiune permisă era mutarea câte unui singur disc de pe o tijă pe alta, astfel încât niciodată să nu se pună un disc mai mare peste un disc mai mic. Legenda spune că sfârşitul lumii se va petrece atunci când călugării vor săvârşi lucrarea. Vom vedea că aceasta se dovedeşte a fi o previziune extrem de optimistă asupra sfârşitului lumii. Presupunând că în fiecare secundă se mută un disc şi se lucrează fără întrerupere, cele 64 de discuri nu pot fi mutate nici în 500 de miliarde de ani de la începutul acţiunii!

Pentru a rezolva problema, vom numerota cele trei tije cu 1, 2 şi respectiv 3. Se observă că pentru a muta cele n discuri de pe tija cu numărul i pe tija cu numărul j (i şi j iau valori între 1 şi 3) este necesar să transferăm primele $n-1$ discuri de pe tija i pe tija $6-i-j$ (adică pe tija rămasă liberă), apoi să transferăm discul n de pe tija i pe tija j, iar apoi retransferăm cele $n-1$ discuri de pe tija $6-i-j$ pe tija j. Cu alte cuvinte, reducem problema mutării a n discuri la problema mutării a $n-1$ discuri. Următoarea metodă Java descrie acest algoritm recursiv.

Listing 9.1: Metoda `hanoi`

```java
public static void hanoi(int n, int i, int j)
{
    if (n > 0)
    {
        hanoi(n - 1, i, 6 - i - j);
        System.out.println(i + "-->" + j);
        hanoi(n - 1, 6 - i - j, j);
    }
}
```

Pentru rezolvarea problemei iniţiale, facem apelul

```java
hanoi(64, 1, 2);
```

Considerăm instrucţiunea `println` ca barometru. Timpul necesar algoritmului este exprimat prin următoare recurenţă:

$$t(n) = \begin{cases} 1 & \text{dacă} \quad n = 1 \\ 2t(n-1) + 1 & \text{dacă} \quad n > 1 \end{cases}$$

Vom demonstra că $t(n) = 2^n - 1$. Rezultă că $t \in \Theta(2^n)$.

Acest algoritm este optim în sensul că este imposibil să mutăm discuri de pe o tijă pe alta cu mai puţin de $2^n - 1$ operaţii. Pentru a muta 64 de discuri vor fi în consecinţă necesare un număr astronomic de 2^{64} operaţii. Implementarea în oricare limbaj de programare care admite exprimarea recursivă se poate face aproape în mod direct.

9.4 Analiza algoritmilor recursivi

Am văzut în exemplul precedent cât de puternică şi în acelaşi timp cât de elegantă este recursivitatea în elaborarea unui algoritm. Cel mai important câştig al exprimării recursive este faptul că ea este naturală şi compactă, fără să ascundă esenţa algoritmului prin detaliile de implementare. Pe de altă parte, apelurile recursive trebuie folosite cu discernământ, deoarece solicită şi ele resursele calculatorului (timp şi memorie). Analiza unui algoritm recursiv implică aproape întotdeauna rezolvarea unui sistem de recurenţe. Vom vedea în continuare cum pot fi rezolvate astfel de recurenţe. Începem cu tehnica cea mai simplă.

9.4.1 Metoda iteraţiei

Cu puţină experienţă şi intuiţie putem rezolva de multe ori astfel de recurenţe prin *metoda iteraţiei*: se execută primii paşi, se intuieşte forma generală, iar apoi se demonstrează prin inducţie matematică că forma este corectă. Să considerăm de exemplu recurenţa problemei turnurilor din Hanoi. Se observă că pentru a muta n discuri este necesar să mutăm $n - 1$ discuri, apoi să mutăm un disc şi în final din nou $n - 1$ discuri. În consecinţă, pentru un anumit $n > 1$ obţinem succesiv:

$$t(n) = 2t(n-1) + 1 = 2^2 t(n-2) + 2 + 1 = \ldots = 2^{n-1} t(1) + \sum_{i=0}^{n-2} 2^i$$

Rezultă că $t(n) = 2^n - 1$. Prin inducţie matematică se demonstrează acum cu uşurinţă că această formă generală este corectă.

9.4.2 Inducţia constructivă

Inducţia matematică este folosită de obicei ca tehnică de demonstrare a unei aserţiuni deja enunţate. Vom vedea în această secţiune că inducţia matematică poate fi utilizată cu succes şi în descoperirea parţială a enunţului aserţiunii. Aplicând această tehnică, putem simultan să demonstrăm o aserţiune doar parţial specificată şi să descoperim specificaţiile care lipsesc şi datorită cărora aserţiunea este corectă. Vom vedea că această tehnică a *inducţiei constructive* este utilă pentru rezolvarea anumitor recurenţe care apar în contextul analizei algoritmilor. Începem cu un exemplu.

Fie funcţia $f : \mathbf{N} \to \mathbf{N}$ definită prin recurenţa:

$$f(n) = \begin{cases} 0 & \text{dacă} \quad n = 1 \\ f(n-1) + n & \text{altfel} \end{cases}$$

Să presupunem pentru moment că nu ştim că $f(n) = \frac{n(n+1)}{2}$. Avem

$$f(n) = \sum_{i=0}^{n} i \leq \sum_{i=0}^{n} n = n^2$$

şi deci $f(n) \in O(n^2)$. Aceasta ne sugerează să formulăm *ipoteza inducţiei specificate parţial IISP(n)* conform căreia f este de forma $f(n) = an^2 + bn + c$. Această ipoteză este parţială în sensul că a, b şi c nu sunt încă cunoscute. Tehnica inducţiei constructive constă în a demonstra prin inducţie matematică această ipoteză incompletă şi a determina în acelaşi timp valorile constantelor necunoscute a, b şi c.

Presupunem că *IISP(n-1)* este adevărată pentru un anumit $n \geq 1$. Atunci, $f(n-1) = a(n-1)^2 + b(n-1) + c = an^2 + (1+b-2a)n + (a-b+c)$. Dacă dorim să arătăm că *IISP(n)* este adevărată, trebuie să arătăm că $f(n) = an^2 + bn + c$. Prin identificarea coeficienţilor puterilor lui n, obţinem ecuaţiile $1 + b - 2a = b$ şi $a - b + c = c$, cu soluţia $a = b = \frac{1}{2}$, c putând fi oarecare. Avem acum o ipoteză "mai completă" (abuzul de limbaj este inevitabil), pe care o numim tot *IISP(n)*, $f(n) = \frac{n^2}{2} + \frac{n}{2} + c$. Am arătat că dacă *IISP(n-1)* este adevărată pentru un anumit $n \geq 1$, atunci este adevărată şi *IISP(n)*. Rămâne să arătăm că este adevărată şi *IISP(0)*. Trebuie să arătăm că $f(0) = a0^2 + b0 + c$. Ştim că $f(0) = 0$, deci *IISP(0)* este adevărată pentru $c = 0$. În concluzie am demonstrat că $f(n) = \frac{n^2}{2} + \frac{n}{2}$ pentru orice n.

9.4.3 Recurenţe liniare omogene

Există din fericire şi tehnici care pot fi folosite aproape automat pentru a rezolva anumite clase de recurenţe. Vom începe prin a considera ecuaţii recurente liniare omogene, adică ecuaţii de forma:

$$a_0t_n + a_1t_{n-1} + \ldots + a_kt_{n-k} = 0 \qquad (*)$$

unde t_i sunt valorile pe care le căutăm, iar coeficienţii a_i sunt constante.

Conform intuiţiei[2], vom căuta soluţii de forma:

$$t_n = x^n$$

unde x este o constantă (deocamdată necunoscută). Dacă înlocuim această soluţie în ecuaţia (*), obţinem

$$a_0x^n + a_1x^{n-1} + \ldots + a_kx^{n-k} = 0$$

Soluţiile acestei ecuaţii sunt fie soluţia trivială $x = 0$, care nu ne interesează, fie soluţiile ecuaţiei:

$$a_0x^k + a_1x^{k-1} + \ldots + a_k = 0$$

care se numeşte *ecuaţia caracteristică* a recurenţei liniare şi omogene (*).

Presupunând deocamdată că cele k rădăcini $r_1, r_2, \ldots, r_k$ ale acestei ecuaţii caracteristice sunt distincte, se verifică uşor că orice combinaţie liniară

$$t_n = \sum_{i=1}^{k} c_i r_i^n$$

este o soluţie a recurenţei (*), unde constantele $c_1, c_2, \ldots, c_k$ sunt determinate de condiţiile iniţiale. Se poate demonstra faptul că (*) are soluţii *numai* de această formă.

Să exemplificăm prin recurenţa care defineşte şirul lui Fibonacci

$$t_n = t_{n-1} + t_{n-2}, n \geq 2$$

iar $t_0 = 0, t_1 = 1$. Putem să rescriem această recurenţă sub forma

$$t_n - t_{n-1} - t_{n-2} = 0$$

care are ecuaţia caracteristică

$$x^2 - x - 1 = 0$$

cu rădăcinile $r_{1,2} = \frac{1 \mp \sqrt{5}}{2}$. Soluţia generală are forma:

$$t_n = c_1r_1^n + c_2r_2^n$$

[2]De fapt, adevărul este că aici nu este vorba de intuiţie, ci de experienţă.

Impunând condiţiile iniţiale, $t_0 = 0, t_1 = 1$, obţinem

$$c_1 + c_2 = 0, n = 0$$

$$c_1 r_1 + c_2 r_2 = 1, n = 1$$

de unde determinăm

$$c_{1,2} = \mp \frac{1}{\sqrt{5}}$$

Deci, $t_n = \frac{1}{\sqrt{5}}(r_1^n + r_2^n)$. Observăm că $r_1 = \phi, r_2 = -\phi^{-1}$ şi obţinem:

$$t_n = \frac{1}{\sqrt{5}}(\phi^n - (-\phi)^{-n})$$

care este cunoscuta relaţie a lui Moivre, descoperită la începutul secolului XVIII. Nu prezintă nici o dificultate să arătăm acum că timpul pentru calculul recursiv al şirului lui Fibonacci este în $\Theta(\phi^n)$.

Cum procedăm însă atunci când rădăcinile ecuaţiei caracteristice nu sunt distincte? Se poate arăta că dacă r este o rădăcină de multiplicitate m a ecuaţiei caracteristice, atunci $t_n = r^n, t_n = nr^n, t_n = n^2 r^n, \ldots, t_n = n^{m-1} r^n$ sunt soluţii pentru `(*)`. Soluţia generală pentru o astfel de recurenţă este atunci o combinaţie liniară a acestor termeni şi a termenilor proveniţi de la celelalte rădăcini ale ecuaţiei caracteristice. Din nou, sunt de determinat exact k constante din condiţiile iniţiale.

Vom da din nou un exemplu. Fie recurenţa

$$t_n = 5t_{n-1} - 8t_{n-2} + 4t_{n-3}$$

cu $t_0 = 0, t_1 = 1, t_2 = 2$. Ecuaţia caracteristică are rădăcinile 1 (de multiplicitate 1) şi 2 (de multiplicitate 2). Soluţia generală este:

$$t_n = c_1 1^n + c_2 2^n + c_3 n 2^n$$

Din condiţiile iniţiale, obţinem $c_1 = -2, c_2 = 2, c_3 = -\frac{1}{2}$.

9.4.4 Recurenţe liniare neomogene

Considerăm acum recurenţe de următoarea formă mai generală

$$a_0 t_n + a_1 t_{n-1} + \ldots a_k t_{n-k} = b^n p(n) \quad (**)$$

unde *b* este o constantă, iar *p(n)* este un polinom în *n* de grad *d*. Ideea generală este ca prin manipulări convenabile să reducem un astfel de caz la o formă omogenă.

De exemplu, o astfel de recurenţă poate fi:

$$t_n - 2t_{n-1} = 3^n$$

În acest caz $b = 3$ şi $p(n) = 1$ un polinom de grad 0. O simplă manipulare ne permite să reducem acest exemplu la forma `(*)`. Înmulţind recurenţa cu 3, obţinem:

$$3t_n - 6t_{n-1} = 3^{n+1}$$

Înlocuind pe *n* cu $n + 1$ în recurenţa originală, avem:

$$t_{n+1} - 2t_n = 3^{n+1}$$

În final, scădem aceste două ecuaţii şi obţinem:

$$t_{n+1} - 5t_n + 6t_{n-1} = 0$$

Am obţinut o recurenţă omogenă pe care o putem rezolva ca în secţiunea precedentă. Ecuaţia caracteristică este:

$$x^2 - 5x + 6 = 0$$

adică $(x - 2)(x - 3) = 0$.

Intuitiv, observăm că factorul $(x - 2)$ corespunde părţii stângi a recurenţei originale, în timp ce factorul $(x - 3)$ a apărut ca rezultat al manipulărilor efectuate pentru a scăpa de partea dreaptă.

Iată un al doilea exemplu:

$$t_n - 2t_{n-1} = (n + 5)3^n$$

Manipulările necesare sunt puţin mai complicate. Trebuie să:

1. Înmulţim recurenţa cu 9;
2. Înlocuim în recurenţă pe *n* cu $n + 2$;
3. Înlocuim în recurenţă pe *n* cu $n + 1$ şi să înmulţim apoi cu -6.

Adunând cele trei ecuaţii obţinute anterior avem:

$$t_{n+2} - 8t_{n+1} + 21t_n - 18t_{n-1} = 0$$

Am ajuns din nou la o ecuaţie omogenă. Ecuaţia caracteristică corespunzătoare este

$$x^3 - 8x^2 + 21x - 18 = 0$$

adică $(x-2)(x-3)^2$.

Încă o dată, observăm că factorul $(x-2)$ provine din partea stângă a recurenţei originale, în timp ce factorul $(x-3)^2$ este rezultatul manipulării.

Generalizând acest procedeu, se poate arăta că pentru a rezolva `(**)` este suficient să luăm următoarea ecuaţie caracteristică:

$$(a_0 x^k + a_1 x^{k-1} + \ldots + a_k)(x-b)^{d+1} = 0$$

Odată ce s-a obţinut această ecuaţie, se procedează ca în cazul omogen.

Vom rezolva acum recurenţa corespunzătoare problemei turnurilor din Hanoi

$$t_n = 2t_{n-1} + 1, n \geq 1$$

iar $t_0 = 0$. Rescriem recurenţa astfel

$$t_n - 2t_{n-1} = 1$$

care este de forma `(**)` cu $b = 1$ şi $p(n) = 1$, un polinom cu grad 0. Ecuaţia caracteristică este atunci $(x-1)(x-2)$, cu soluţiile 1 şi 2. Soluţia generală a recurenţei este:

$$t_n = c_1 1^n + c_2 2^n$$

Avem nevoie de două condiţii iniţiale. Ştim că $t_0 = 0$; pentru a găsi cea de-a doua condiţie calculăm

$$t_1 = 2t_0 + 1$$

Din condiţiile iniţiale, obţinem $t_n = 2^n - 1$.

Observaţie: dacă ne interesează doar ordinul lui t_n, nu este necesar să calculăm efectiv constantele în soluţia generală. Dacă ştim că $t_n = c_1 1^n + c_2 2^n$, rezultă că $t_n \in O(2^n)$. Din faptul că numărul de mutări a unor discuri nu poate fi negativ sau constant (deoarece avem în mod evident $t_n \geq n$), deducem că $c_2 > 0$. Avem atunci $t_n \in \Omega(2^n)$ şi deci $t_n \in \Theta(2^n)$. Putem obţine chiar ceva mai mult.

Substituind soluţia generală înapoi în recurenţa originară, găsim

$$1 = t_n - 2t_{n-1} = c_1 + c_2 2^n - 2(c_1 + c_2 2^{n-1}) = -c_1$$

Indiferent de condiţia iniţială, c_1 este -1.

9.4.5 Schimbarea variabilei

Uneori putem rezolva recurenţe mai complicate printr-o schimbare de variabilă. În exemplele care urmează, vom nota cu *T(n)* termenul general al recurenţei şi cu t_k termenul noii recurenţe obţinute printr-o schimbare de variabilă. Presupunem pentru început că *n* este o putere a lui 2.

Un prim exemplu este recurenţa

$$T(n) = 4T(\frac{n}{2}) + n, n > 1$$

în care înlocuim pe *n* cu 2^k, notăm $t_k = T(2^k) = T(n)$ şi obţinem:

$$t_k = 4t_{k-1} + 2^k$$

Ecuaţia caracteristică a acestei recurenţe liniare este (conform paragrafului anterior):

$$(x - 4)(x - 2) = 0$$

şi deci $t_k = c_1 4^k + c_2 2^k$. Înlocuim pe *k* cu *lg n*

$$T(n) = c_1 n^2 + c_2 n$$

Rezultă că $T(n) \in O(n^2 \mid n\, este\, o\, putere\, a\, lui\, 2)$.

Un al doilea exemplu îl reprezintă ecuaţia

$$T(n) = 4T(\frac{n}{2}) + n^2, n > 1$$

Procedând la fel, ajungem la recurenţa

$$t_k = 4t_{k-1} + 4^k$$

cu ecuaţia caracteristică

$$(x - 4)^2 = 0$$

şi soluţia generală $t_k = c_1 4^k + c_2 k 4^k$. Atunci,

$$T(n) = c_1 n^2 + c_2 n^2\, lg\, n$$

şi obţinem că $T(n) \in O(n^2\, log\, n \mid n\, este\, o\, putere\, a\, lui\, 2)$.

În fine, să considerăm şi exemplul

$$T(n) = 3T(\tfrac{n}{2}) + cn, n > 1$$

c fiind o constantă. Obţinem succesiv

$$T(2^k) = 3T(2^{k-1}) + c2^k$$

$$t_k = 3t_{k-1} + c2^k$$

cu ecuaţia caracteristică

$$(x-3)(x-2) = 0$$

$$t_k = c_1 3^k + c_2 2^k$$

$$T(n) = c_1 3^{lg\, n} + c_2 n$$

şi, deoarece

$$a^{lg\, b} = b^{lg\, a}$$

obţinem

$$T(n) = c_1 n^{lg\, 3} + c_2 n$$

deci, $T(n) \in O(n^{lg\, 3} \mid n\, este\, o\, putere\, a\, lui\, 2)$.

În toate aceste exemple am folosit notaţia asimptotică condiţionată. Pentru a arăta că rezultatele obţinute sunt adevărate pentru orice n, este suficient să adăugăm condiţia ca $T(n)$ să fie crescătoare pentru $n > n_0$.

Putem enunţa acum o proprietate care este utilă ca reţetă pentru analiza algoritmilor cu recursivităţi de forma celor din exemplele precedente. Proprietatea, a cărei demonstrare o lăsăm ca exerciţiu, este foarte utilă la analiza algoritmilor Divide et Impera.

Propoziţie. Fie $T : \mathbf{N} \to \Re^+$ o funcţie nedescrescătoare

$$T(n) = aT(\tfrac{n}{b}) + cn^k, n > n_0$$

unde $n_0 \geq 1, b \geq 2$ şi $k \geq 0$ sunt întregi, a şi c sunt numere reale pozitive, iar $\frac{n}{n_0}$ este o putere a lui b. Atunci avem:

$$T(n) \in \begin{cases} \Theta(n^k) & \text{dacă} \quad a < b^k \\ \Theta(n^k log\, n) & \text{dacă} \quad a = b^k \\ \Theta(n^{log_b a}) & \text{dacă} \quad a > b^k \end{cases}$$

9.5 Implementarea algoritmilor

Am considerat utilă prezenţa la sfârşitul capitolului a implementărilor Java pentru problemele prezentate pe parcursul acestui capitol. Este cazul algoritmilor de sortare prin selecţie, a celui de sortare prin inserţie şi a algoritmului turnurilor din Hanoi.

Listing 9.2: Implementarea algoritmului de sortare prin selecţia minimului

```
import java.io.*;
import io.Reader;

/**
 * Sortare prin selectia minimului.
 */
public class SortareSelMin
{

  /** Ordonarea prin selectia minimului.*/
  public static void ordonare(int[] a)
  {
    for (int i = 0; i < a.length - 1; i++)
    {
      int pozMin = i;

      for (int j = i + 1; j < a.length; j++)
      {
        if (a[j] < a[pozMin])
        {
          pozMin = j;
        }
      }

      int aux = a[i];
      a[i] = a[pozMin];
      a[pozMin] = aux;
    }
  }

  /** Programul principal.*/
  public static void main(String[] args)
  {
    //citirea elementelor sirului
    System.out.println("Introduceti elementele sirului " +
      " (pe aceeasi linie, separate prin spatiu):");
    int[] s = Reader.readIntArray();

    //ordonarea sirului s
    ordonare(s);
```

```

    //afisare rezultate
    System.out.print("Sirul ordonat este: ");
    for (int i = 0; i < s.length; i++)
    {
      System.out.print(s[i] + " ");
    }
    System.out.println();
  }
}
```

Listing 9.3: Implementarea algoritmului de sortare prin inserţie

```
import java.io.*;
import io.Reader;

/**
 * Sortare prin insertie.
 */
public class SortareInsertie
{

  /** Ordonarea prin insertie.*/
  public static void ordonare(int[] a)
  {
    for (int i = 1; i < a.length; i++)
    {
      int tmp = a[i];
      int j = i;

      for (; (j > 0) && (tmp < a[j - 1]); j--)
      {
        a[j] = a[j - 1];
      }

      a[j] = tmp;
    }
  }

  /** Programul principal.*/
  public static void main(String[] args)
  {
    //citirea elementelor sirului
    System.out.println("Introduceti elementele sirului" +
      " (pe aceeasi linie, separate prin spatiu):");
    int[] s = Reader.readIntArray();

    //ordonarea sirului s
    ordonare(s);

```

```
    //afisare rezultate
    System.out.print("Sirul ordonat este: ");
    for (int i = 0; i < s.length; i++)
    {
      System.out.print(s[i] + " ");
    }
    System.out.println();
  }
}
```

Listing 9.4: Implementarea problemei turnurilor din Hanoi

```
import java.io.*;
import io.Reader;

/**
 * Turnurile din Hanoi.
 */
public class Hanoi
{
  /** Implementarea algoritmului "Turnurile din Hanoi".*/
  public static void hanoi(int n, int i, int j)
  {
    if (n > 0)
    {
      hanoi(n - 1, i, 6 - i - j);
      System.out.println(i + " --> " + j);
      hanoi(n - 1, 6 - i - j, j);
    }
  }

  /** Programul principal.*/
  public static void main(String[] args)
  {
    //citirea numarului de discuri
    System.out.print("Introduceti numarul de discuri: ");

    int n = Reader.readInt();

    //apelul metodei hanoi
    hanoi(n, 1, 2);
  }
}
```

Rezumat

Capitolul de faţă a realizat o scurtă introducere în domeniul vast al analizei algoritmilor. Ideea cea mai importantă care reiese de aici este că algoritmii

utilizaţi afectează timpul de execuţie al unui program mult mai drastic decât artificiile de programare. Algoritmii care au un timp de lucru exponenţial nu sunt în general aplicabili pentru date de intrare de dimensiuni rezonabile, spre deosebire de cei liniari sau pătratici al căror timp de execuţie nu creşte atât de drastic odată cu dimensiunea datelor de intrare.

Capitolul următor prezintă cele mai importante structuri de date, împreună cu gradul lor de eficienţă şi cu operaţiile pe care le permit. De asemenea, sunt prezentate diverse situaţii în care pot fi folosite structurile de date. Restul capitolelor sunt dedicate prezentării algoritmilor fundamentali.

Noţiuni fundamentale

$O(f)$: notaţie utilizată pentru a determina termenul dominant al funcţiei `f`. Cu ajutorul ei se limitează superior timpul de execuţie al unui algoritm.

$\Omega(f)$: notaţie utilizată pentru a limita inferior timpul de execuţie al unui algoritm.

$\Theta(f)$: notaţie utilizată pentru a arăta că timpul de execuţie al unui algoritm este limitat atât inferior cât şi superior de câte un multiplu real al aceleaşi funcţii f.

algoritm liniar: algoritm care are timpul de execuţie $O(n)$.

algoritm exponenţial: algoritm al cărui timp de execuţie creşte exponenţial în raport cu dimensiunea datelor de intrare ($O(a^n)$).

algoritm polinomial: algoritm al cărui timp de execuţie este mărginit superior de un polinom ($O(n^k)$).

ecuaţie caracteristică: ecuaţie polinomială ataşată recurenţelor liniare, prin a cărei rezolvare se găseşte soluţia recurenţei.

inducţia constructivă: o variantă a inducţiei matematice care permite demonstrarea unei aserţiuni parţial sau incomplet enunţate precum şi descoperirea specificaţiilor care lipsesc din aserţiunea respectivă.

metoda iteraţiei: metodă simplă de rezolvare a recurenţelor liniare în care se execută primii paşi ai recurenţei după care se intuieşte forma generală care se demonstrează prin inducţie.

recurenţă liniară: formulă de recurenţă în care termenul al n-lea este exprimat ca o combinaţie liniară a termenilor precedenţi.

schimbarea variabilei: tehnică utilizată pentru a reduce anumite formule de recurenţă la o formă liniară.

Erori frecvente

1. Pentru cicluri iterative imbricate, timpul total de execuţie este produsul dimensiunilor ciclurilor, în timp ce în cazul ciclurilor iterative consecutive, afirmaţia nu este adevărată. De exemplu, pentru două cicluri imbricate care se execută fiecare de la 1 la n^2, timpul total de execuţie este $O(n^4)$.

2. Nu scrieţi expresii de tipul $O(2n^2)$ sau $O(n^2+n)$. În cele mai multe situaţii, este necesar doar termenul dominant, fără constanta care îl precede. În consecinţă notaţia utilizată pentru ambele cazuri de mai înainte este $O(n^2)$.

3. Pentru a exprima limita inferioară a complexităţii unui algoritm, utilizaţi notaţia Ω, nu O.

4. Baza în care se scrie logaritmul este irelevantă pentru notaţiile asimptotice. Astfel, $O(\lg n)$ este egal cu $O(\ln n)$ deoarece ele diferă doar printr-o constantă multiplicativă.

Exerciţii

1. Care din următoarele afirmaţii sunt adevărate?

   (a) $n^2 \in O(n^3)$
   (b) $n^3 \in O(n^2)$
   (c) $2^{n+1} \in O(2^n)$
   (d) $(n+1)! \in O(n!)$
   (e) pentru orice funcţie $f : \mathbf{N} \to \mathbf{R}^*$, $f \in O(n) \Rightarrow [f^2 \in O(n^2)]$
   (f) pentru orice funcţie $f : \mathbf{N} \to \mathbf{R}^*$, $f \in O(n) \Rightarrow [2^f \in O(2^n)]$

2. Demonstraţi că relaţia "$\in O$" este *tranzitivă*: dacă $f \in O(g)$ şi $g \in O(h)$, atunci $f \in O(h)$. Deduceţi de aici că dacă $g \in O(h)$, atunci $O(g) \subseteq O(h)$.

3. Găsiţi două funcţii $f, g : \mathbf{N} \to \mathbf{R}^*$, astfel încât $f \notin O(g)$ şi $g \notin O(f)$.

   Soluţie: $f(n) = n, g(n) = n^{1+sin\, n}$.

4. Pentru oricare două funcţii $f, g : \mathbf{N} \to \mathbf{R}^*$ definim următoarea relaţie binară: $f \leq g$ dacă $O(f) \subseteq O(g)$. Demonstraţi că relaţia "$\leq$" este o relaţie de ordine parţială în mulţimea funcţiilor definite pe **N** şi cu valori în $\mathbf{R}^*$.

   Indicaţie: Trebuie arătat că relaţia este reflexivă, tranzitivă şi antisimetrică. Ţineţi cont de exerciţiul 3.

5. Pentru oricare două funcţii $f, g : \mathbf{N} \to \mathbf{R}^*$demonstraţi că $O(f + g) = O(max(f, g))$ unde suma şi maximul se iau punctual.

6. Fie $f(n) = a_m n^m + \ldots + a_1 n + a_0$ un polinom de grad m, cu $a_m > 0$. Arătaţi că $f \in O(n^m)$.

7. Considerăm afirmaţia $O(n^2) = O(n^3 + (n^2 - n^3)) = O(max(n^3, n^2 - n^3)) = O(n^3)$. Unde este eroarea?

8. Considerăm afirmaţia $\sum_{i=1}^{n} i = 1 + 2 + \ldots + n \in O(1 + 2 + \ldots + n) = O(max(1 + 2 + \ldots + n)) = O(n)$. Unde este eroarea?

9. Pentru oricare două funcţii $f, g : \mathbf{N} \to \mathbf{R}^*$ demonstraţi că $\Theta(f) + \Theta(g) = \Theta(f + g) = \Theta(max(f, g)) = max(\Theta(f), \Theta(g))$, unde suma şi maximul se iau punctual.

10. Analizaţi eficienţa următorilor algoritmi:

    (a)
    ```
    pentru i=1,n
        pentru j=1,5
            {operaţie elementară}
    ```
    (b)
    ```
    pentru i=1,n
        pentru j=1,i+1
            {operaţie elementară}
    ```
    (c)
    ```
    pentru i=1,n
        pentru j=1,6
            pentru k=1,n
                {operaţie elementară}
    ```
    (d)
    ```
    pentru i=1,n
        pentru j=1,i
            pentru k=1,n
                {operaţie elementară}
    ```

11. Construiţi un algoritm cu timpul în $\Theta(n \log n)$.

12. Fie un algoritm:
```
pentru i=0,n
        j←i
        cât timp j<>0
                j←j div 2
```
Găsiţi ordinul exact al timpului de execuţie.

13. Rezolvaţi următoarea recurenţă: $t_n - 3t_{n-1} - 4t_{n-2} = 0, n \geq 2$ cu $t_0 = 0, t_1 = 1$.

14. Care este timpul de execuţie pentru un algoritm recursiv cu recurenţa: $t_n = 2t_{n-1} + n$?

    Indicaţie: Se ajunge la ecuaţia caracteristică $(x-2)(x-1)^2 = 0$, *iar soluţia generală este* $t_n = c_1 2^n + c_2 1^n + c_3 n 1^n$. *Rezultă că* $t_n \in O(2^n)$. *Substituind soluţia generală în recurenţă, obţinem că, indiferent de condiţia iniţială,* $c_2 = -2$ *şi* $c_3 = -1$. *Atunci, toate soluţiile interesante ale recurenţei trebuie să aibă* $c_1 > 0$ *şi ele sunt toate în* $\Omega(2^n)$, *deci în* $\Theta(2^n)$.

15. Să se calculeze secvenţa de sumă maximă, formată din termeni consecutivi, ai unui şir de numere.

10. Structuri de date

Nu poţi obţine întotdeauna ceea ce doreşti, dar, dacă încerci, uneori vei obţine ceea ce ai nevoie.

Autor anonim

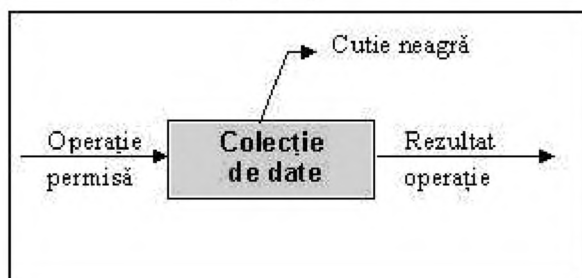
Două programe care rezolvă aceeaşi problemă pot să arate complet diferit. Unul poate fi extrem de lizibil, concis şi uşor de modificat pentru a se adapta la rezolvarea unor probleme asemănătoare, iar celălalt poate fi impenetrabil, obscur, interminabil şi dificil de modificat. Cele două programe pot să difere atât de mult în ceea ce priveşte durata de execuţie şi necesarul de memorie încât, pentru un anumit set de intrare, unul furnizează răspunsul după 2 secunde, iar celălalt după câteva secole!

Experienţa a demonstrat că aceste diferenţe sunt generate de structura programului şi de structura datelor.

Programele sunt scrise pentru a rezolva probleme reale. Structurarea programului împarte problema şi soluţia ei în componente mai simple şi mai uşor de înţeles. Informaţia care trebuie procesată este reţinută în structuri de date (tablouri, înregistrări, liste, stive, arbori, fişiere etc.). O structură de date grupează datele. O structură de date aleasă adecvat poate face operaţiile simple şi eficiente, iar una aleasă neadecvat poate face operaţiile alambicate şi ineficiente.

Structurile de date conţin informaţie asupra căreia se operează în timpul execuţiei unui program. Despre programe obişnuim să spunem că procesează informaţie, când, de fapt, ele procesează structuri de date. Astfel, nu mai pare surprinzător faptul că structura datelor şi a programelor sunt extrem de importante şi că ele trebuie corelate corespunzător pentru a programa cu succes. Asimilarea cunoştinţelor legate de structuri de date şi algoritmi, alături de stăpânirea conceptelor fundamentale ale programării orientate pe obiecte, îi permit programatorului să menţină o supremaţie regală asupra programelor, astfel încât

Figura 10.1: Închiderea datelor într-o cutie neagră. Datele pot fi accesate doar prin invocarea unei operaţii permise.



acestea să rămână lizibile, uşor de întreţinut şi eficiente, chiar şi atunci când cresc în dimensiune.

Atât programele cât şi datele au o anumită structurare şi atât structura programului cât şi a datelor trebuie să fie adecvată problemei de rezolvat. Vom urmări să prezentăm structurile de date nu ca un subiect teoretic izolat, ci ca pe un instrument esenţial al procesului de rezolvare a problemelor care conduce la crearea de programe eficiente.

Structurile de date sunt importante, deoarece modul în care programatorul alege să reprezinte datele afectează în mod semnificativ claritatea, concizia, viteza de execuţie şi necesarul de memorie al programului. În acest capitol, cât şi în cele care urmează, vom prezenta cum să folosim diferite structuri de date pentru a crea programe corecte şi eficiente. Vom vedea că operaţiile care trebuie realizate asupra datelor sunt cele care determină care este cea mai potrivită structură de date care trebuie folosită.

Dezvoltarea de programe este dificilă, mai ales atunci când este făcută fără nici un fel de strategie. Progamarea orientată pe obiecte, prezentată în prima parte a lucrării, uşurează mult dezvoltarea programelor şi conduce la programe bine structurate. Ea constă în împărţirea atentă a unei probleme complexe în componente mai simple a căror interfaţă este apoi cu uşurinţă combinată pentru a rezolva problema iniţială.

Abstractizarea datelor constă în a trata o colecţie de date extrăgând aspectele sale esenţiale, ignorând pe cât se poate detaliile. Abstractizarea datelor reduce datele la o colecţie şi la operaţiile care se pot realiza asupra acestei colecţii. Efectul este ca şi cum colecţia de date ar fi închisă într-o cutie neagră (black box) impenetrabilă, singurul mod de a accesa datele fiind invocarea uneia sau mai multor operaţii permise (**Figura** 10.1).

Modul în care datele sunt aşezate în acea cutie neagră şi modul în care operaţiile se execută devin detalii irelevante[1]. Astfel de detalii determină eficienţa programului, dar nu îi afectează structura logică.

Abstractizarea datelor trebuie privită mai degrabă ca o facilitate care uşurează efortul de programare, şi nu ca o nouă constrângere în privinţa stilului de programare. Pentru a înţelege mai bine aceste noţiuni, să vedem cum poate fi definită simplificat o stivă ca tip abstract de date. Stiva este o colecţie de date omogene (de acelaşi tip) asupra căreia se pot realiza următoarele operaţii:

- PUSH(X) - are ca efect depunerea lui X pe vârful stivei;
- POP(X) - are ca efect încărcarea valorii din vârful stivei în parametrul X şi eliminarea vârfului stivei.

Modul în care cele două operaţii (PUSH şi POP) sunt implementate şi modul în care datele sunt reţinute în stivă (static, înlănţuit etc.) nu trebuie să transpară utilizatorului; pe el pur şi simplu nu trebuie să îl intereseze acest lucru. Este exact ca şi curentul electric: atunci când acţionăm comutatorul de la veioză, ştim că becul se va aprinde; care este procesul prin care filamentul becului se încinge şi emite lumină nu ne priveşte. Tot astfel, şi listele, cozile, arborii binari împreună cu operaţiile care se fac asupra lor pot fi privite ca tipuri abstracte de date.

Unui tip abstract de date *mulţime* putem să-i asociem operaţii cum ar fi reuniune, intersecţie, dimensiune, complementară. Într-o altă situaţie, putem avea nevoie numai de operatorii de reuniune şi apartenenţă, care definesc un alt tip abstract de date (TAD), care poate să aibă o cu totul altă organizare internă, deoarece, aşa cum am spus, operaţiile sunt cele care definesc TAD-ul.

În consecinţă, rolul acestui capitol este de a prezenta modul în care se construiesc aceste tipuri abstracte de date (sau structuri de date) pentru a le putea utiliza apoi în crearea de programe robuste, lizibile şi eficiente. Ideea de bază este că implementarea operaţiilor unui TAD se realizează o singură dată în cadrul unei aplicaţii, şi orice altă parte a aplicaţiei va utiliza structura de date prin itermediul interfeţei pe care aceasta o expune.

Dacă din diverse motive anumite detalii de implementare trebuie schimbate, acest lucru se va face uşor prin modificarea rutinelor care realizează operaţiile din TAD. Aceste schimbări nu vor afecta în nici un fel restul programului, deoarece interfaţa se menţine neschimbată. În exemplul nostru cu stiva, dacă decidem să trecem de la alocarea secvenţială (statică) a elementelor stivei la

[1] Irelevante pentru cel care utilizează structura de date. Noi ne vom ocupa în acest capitol tocmai de modul de construcţie al acestei cutii negre, pentru a o putea utiliza apoi ori de câte ori este necesar.

alocarea înlănţuită (dinamică), implementarea operaţiilor PUSH şi POP va trebui în mod cert schimbată; această schimbare nu va afecta în nici un fel restul programului, care va utiliza operaţiile PUSH şi POP expuse de stivă fără a sesiza că implementarea acestora este diferită.

Acest capitol prezintă şase dintre cele mai cunoscute structuri de date: stive, cozi, liste înlănţuite, arbori binari de căutare, tabele de repartizare (engl. hashtables) şi cozi de prioritate. Scopul este acela de a defini fiecare structură de date şi de a furniza o estimare intuitivă pentru complexitatea operaţiilor de inserare, ştergere şi acces. Implementarea operaţiilor va fi făcută pentru fiecare structură de date separat.

În acest capitol vom vedea:

- Descrierea structurilor de date uzuale, operaţiile permise pe ele şi timpii lor de execuţie;
- Pentru fiecare structură de date, vom defini o interfaţă Java conţinând protocolul care trebuie implementat;
- Programele complete pentru implementarea acestor structuri.

Scopul este acela de a arăta că specificarea, care descrie funcţionalitatea, este independentă de implementare. Nu trebuie să ştim *cum* este implementat un anumit lucru, atâta timp cât ştim că *este* implementat.

10.1 Cum implementăm structurile de date?

Am arătat deja că structurile de date ne permit atingerea unui scop important în programarea orientată pe obiecte: *reutilizarea componentelor*. Aşa cum vom vedea mai târziu în acest capitol, structurile de date descrise sunt folosite în multe situaţii. Odată ce o structură de date a fost implementată, ea poate fi folosită din nou şi din nou în aplicaţii de natură diversă[2].

Această abordare - separarea interfeţei de implementare - este o parte fundamentală a orientării pe obiecte. Cel care foloseşte structura de date nu trebuie să vadă implementarea ei, ci doar operaţiile admisibile. Aceasta ţine de partea de ascundere a informaţiei din programarea orientată pe obiecte. O altă parte importantă a programării orientate pe obiecte este *abstractizarea*. Trebuie să proiectăm cu grijă structura de date, deoarece vom scrie programe care folosesc aceste structuri de date fără să aibă acces la implementarea lor. Aceasta va face

[2]De altfel, bibliotecile Java cuprind majoritatea structurilor de date uzuale prezentate în acest capitol: Stack, Queue, Hashtable etc.

Listing 10.1: Interfaţa `IMemoryCell` pentru clasa `MemoryCell`

```
/** Interfata abstracta pentru o celula de memorie care
* stocheaza un obiect de tip arbitrar */
public interface IMemoryCell
{
  /** Intoarce elementul stocat in cadrul celulei*/
  Object read();
  /** Scrie obiectul x in celula*/
  void write(Object x);
}
```

în schimb ca interfaţa să fie mai curată, mai flexibilă, şi, de obicei, mai uşor de implementat.

Toate structurile de date sunt uşor de implementat dacă nu ne punem problema eficienţei. Acest lucru permite să adăugăm componente "ieftine" în program doar pentru depanare. Putem apoi înlocui aceste implementări "ieftine" cu implementări care au o performanţă (în timp şi/sau în spaţiu) mai bună şi care sunt adecvate pentru procesarea unei cantităţi mai mari de informaţie. Deoarece interfeţele sunt fixate, aceste înlocuiri nu necesită practic nici o modificare în programele care folosesc aceste structuri de date.

Vom descrie structurile de date prin intermediul interfeţelor. De exemplu, stiva este precizată prin intermediul interfeţei `Stack`. Clasa care implementează această interfaţă va implementa toate metodele specificate în `Stack`, la care se mai pot adăuga anumite funcţionalităţi.

Ca un exemplu, în **Listing 10.1** este descrisă o interfaţă pentru clasa `MemoryCell`, utilizată în primul volum, la capitolul *Moştenire*, secţiunea *Implementarea de componente generice*. Interfaţa descrie funcţiile disponibile; clasa concretă trebuie să definească aceste funcţii. Implementarea interfeţei este prezentată în **Listing 10.2**.

Este important de reţinut faptul că structurile de date definite în acest capitol stochează referinţe către elementele inserate, şi nu copii ale elementelor. Am ales această variantă deoarece este bine ca în structura de date să fie plasate obiecte nemodificabile (cum ar fi `String`, `Integer`, etc.) pentru ca un utilizator extern să nu poată să schimbe starea unui obiect care este înglobat într-o structură de date.

Fiecare dintre structurile prezentate este implementată complet, împreună cu un program de test pentru a verifica corectitudinea implementării structurilor de date. Pentru o mai bună organizare, clasele utilizate în aceste programe sunt împărţite pe pachete, ceea ce înseamnă că fiecare fişier sursă este salvat într–un

Listing 10.2: Implementarea clasei `MemoryCell`

```
/** Clasa concreta care implementeaza interfata MemCell */
public class MemoryCell implements MemCell
{
  /** Atribut care indica obiectul stocat */
  private Object storedValue;

  public Object read()
  {
    return storedValue;
  }

  public void write(Object x)
  {
    storedValue = x;
  }
}
```

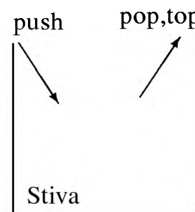
director corespunzător numelui pachetului. Pentru o mai bună înţelegere, considerăm directorul de lucru `c:\javawork` (pentru utilizatorii Windows). Pentru ca programele să funcţioneze, acest director trebuie adăugat la variabila sistem `CLASSPATH`. În acest director se vor crea directoarele corespunzătoare pachetelor existente (în cazul nostru, este vorba de pachetele `datastructures` pentru clasele specifice structurilor de date şi `exceptions` pentru clasele de excepţii), precum şi fişierele sursă ale aplicaţiilor care folosesc aceste structuri de date (clasele corespunzătoare acestor aplicaţii de test nu sunt incluse în pachete).

Unele structuri de date folosesc clase în comun cu alte structuri de date. Din acest motiv, implementările claselor respective nu sunt reluate separat pentru fiecare structură de date.

10.2 Stive

O stivă este o structură de date în care orice tip de acces este permis doar asupra ultimului element inserat. Comportamentul unei stive este foarte asemănător cu cel al unei grămezi de farfurii. Ultima farfurie adăugată va fi plasată în vârf fiind, în consecinţă, uşor de accesat, în timp ce farfuriile puse cu mai mult timp în urmă (aflate sub alte farfurii) vor fi mai greu de accesat, putând periclita stabilitatea întregii grămezi. Astfel, stiva este adecvată în situaţiile în care avem nevoie să accesăm doar elementul din vârf. Toate celelalte elemente

Figura 10.2: Inserarea în stivă se face prin `push()`, accesul prin `top()`, iar ştergerea prin `pop()`.



Listing 10.3: Clasa de excepţii `UnderflowException`

```
package exceptions;

public class UnderflowException extends Exception
{
  public UnderflowException()
  {
    super();
  }

  public UnderflowException(String msg)
  {
    super(msg);
  }
}
```

sunt inaccesibile.

Cele trei operaţii naturale de inserare, ştergere şi căutare, sunt denumite în cazul unei stive, `push`, `pop` şi `top`. Cele trei operaţii sunt ilustrate în **Figura 10.2**, iar o interfaţă Java pentru o stivă abstractă este prezentată în **Listing 10.4**. Interfaţa declară şi o metodă `topAndPop()` care combină două operaţii, consultare şi extragere. În cazul nostru, metodele `pop()`, `top()` şi `topAndPop()` pot arunca o excepţie `UnderflowException` în cazul în care se încearcă accesarea unui element când stiva este goală. Această excepţie va trebui să fie prinsă până la urmă de o metodă apelantă. Clasa `UnderflowException` este definită în **Listing 10.3**, şi ea este practic identică cu clasa `Exception`. Important pentru noi este că diferă *tipul*, ceea ce ne permite să prindem *doar* această excepţie cu o instrucţiune `try-catch`.

Listing 10.6 prezintă un exemplu de utilizare a stivei în care se foloseşte o clasă `StackAr` care reţine elementele stivei într-un şir (array). Clasa `StackAr`

Listing 10.4: Interfaţă pentru o stivă

```
package datastructures;

import exceptions.*;
/** Interfata pentru o stiva. Stiva expune metode pentru
* manipularea (adaugarea, stergerea, consultarea)
* elementului din varful ei*/
public interface Stack
{
  public void push(Object x);

  public void pop() throws UnderflowException;

  public Object top() throws UnderflowException;

  public Object topAndPop() throws UnderflowException;

  public boolean isEmpty();

  public void makeEmpty();
}
```

este o modalitate de implementare a interfeţei `Stack`. `StackAr` foloseşte un şir pentru a reţine elementele din stivă. Mai există şi alte modalităţi de implementare a unei stive. Una dintre acestea se bazează pe liste şi implementarea ei este propusă în exerciţiul 1. **Listing 10.5** prezintă codul sursă al clasei `StackAr`.

Analizând programul de test din **Listing 10.6** se observă că stiva poate fi folosită pentru a inversa ordinea elementelor. De remarcat un mic artificiu folosit aici: ieşirea din ciclul `for` de la liniile 21-24 se realizează în momentul în care stiva se goleşte şi metoda `topAndPop()` aruncă `UnderflowException` care este prinsă la linia 26. Am recurs la acest artificiu pentru a înţelege mecanismul excepţiilor. Folosirea acestui artificiu în mod curent nu este recomandată, deoarece reduce lizibilitatea codului.

Fiecare operaţie pe stivă trebuie să ia o cantitate constantă de timp indiferent de dimensiunea stivei, la fel cum accesarea farfuriei din vârful grămezii este rapidă, indiferent de numărul de farfurii din teanc. Accesul la un element oarecare din stivă nu este eficient, de aceea el nici nu este permis de către interfaţa noastră.

Stiva este deosebit de utilă deoarece sunt multe aplicaţii pentru care trebuie să accesăm doar ultimul element inserat. Un exemplu ilustrativ este salvarea parametrilor şi variabilelor locale în cazul apelului unei alte subrutine.

Implementarea interfeţei `Stack` este realizată folosind şirul de obiecte `elements` (linia 8 în clasa `StackAr`). De aici şi numele de stivă bazată pe şiruri. Practic, elementele stivei sunt păstrate într-un şir de obiecte, pe care se realizează operaţiile expuse de interfaţă: `top()`, `pop()`, `push()` etc. Stiva are o capacitate iniţială de elemente (`DEFAULT_CAPACITY`). Dacă pe măsură ce se fac adăugări în stivă, această capacitate este atinsă, atunci stiva îşi măreşte capacitatea cu un număr precizat de elemente (în cadrul metodei `increaseStackSize()`). Operaţia este însă ascunsă privirilor utilizatorului, pentru că stiva îşi măreşte dimensiunea fără ca utilizatorul să precizeze sau să aibă cunoştinţă de acest lucru. Nivelul de "umplere" al stivei este specificat prin intermediul variabilei `topPosition`, care indică şi poziţia vârfului stivei.

Listing 10.5: Implementarea unei structuri de tip stivă folosind şiruri

```
1 package datastructures;
2
3 import exceptions.*;
4 /** Implementarea unei stive folosind un sir */
5 public class StackAr implements Stack
6 {
7   /** Sir care retine elementele stivei*/
8   private Object[] elements;
9   /**Dimensiunea implicita a stivei*/
10  private static final int DEFAULT_CAPACITY = 10;
11  /** Pozitia varfului stivei */
12  private int topPosition;
13
14  /** Constructor care aloca memorie pentru elements si
15  * initializeaza varful stivei */
16  public StackAr()
17  {
18    elements = new Object[DEFAULT_CAPACITY];
19    topPosition = -1;
20  }
21
22  /** Adauga elementul x in stiva.*/
23  public void push(Object x)
24  {
25    if (topPosition == elements.length - 1)
26    {
27      increaseStackSize();
28    }
29
30    elements[++topPosition] = x;
31  }
32
33  /** Mareste dimensiunea (capacitatea) stivei cu
```

```
* DEFAULT_CAPACITY atunci cand stiva este plina. */
private void increaseStackSize()
{
  Object[] newStack = new Object[elements.length +
  DEFAULT_CAPACITY];

  //copiem elementele in noua stiva
  for (int i = 0; i < elements.length; i++)
  {
    newStack[i] = elements[i];
  }

  //elements devine noua stiva
  elements = newStack;
}

/** Extrage elementul din varful stivei. */
public void pop() throws UnderflowException
{
  if (isEmpty())
  {
    throw new UnderflowException("Stiva vida.");
  }

  topPosition--;
}

/** Returneaza elementul din varful stivei
* (ultimul adaugat).*/
public Object top() throws UnderflowException
{
  if (isEmpty())
  {
    throw new UnderflowException("Stiva vida.");
  }

  return elements[topPosition];
}

/** Returneaza elementul din varful stivei si
* il elimina apoi din stiva.*/
public Object topAndPop() throws UnderflowException
{
  if (isEmpty())
  {
    throw new UnderflowException("Stiva vida.");
  }

  return elements[topPosition--];
}
```

```
  /** Verifica daca stiva e vida. */
  public boolean isEmpty()
  {
    return topPosition == -1;
  }

  /** Elimina toate elementele din stiva. */
  public void makeEmpty()
  {
    topPosition = -1;
  }
}
```

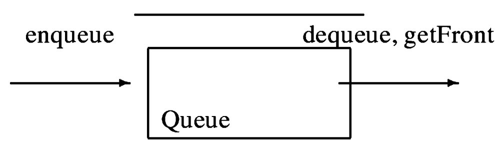
Listing 10.6: Exemplu de utilizare a stivei. Programul va afişa: Conţinutul stivei este 4 3 2 1 0

```
import datastructures.*;
import exceptions.*;
/** Clasa de test simpla pentru o stiva, care adauga 5 numere
* dupa care le extrage in ordine inversa */
public class TestStack
{
  public static void main(String[] args)
  {
    Stack s = new StackAr();

    //introducem elemente in stiva
    for (int i = 0; i < 5; i++)
    {
      s.push(new Integer(i));
    }

    //scoatem elementele din stiva si le afisam
    System.out.print("Continutul stivei este: ");
    try
    {
      for (;;)
      {
        System.out.print(s.topAndPop() + " ");
      }
    }
    catch(UnderflowException ue)
    {
    }
  }
}
```

Figura 10.3: Modelul unei cozi: adăugarea la coadă se face prin `enqueue()`, accesul prin `getFront()`, ştergerea prin `dequeue()`.



10.3 Cozi

O altă structură simplă de date este *coada*. În multe situaţii este important să avem acces şi/sau să ştergem ultimul element inserat. Dar, într-un număr la fel de mare de situaţii, acest lucru nu numai că nu mai este important, este chiar nedorit. De exemplu, într-o reţea de calculatoare care au acces la o singură imprimantă este normal ca dacă în coada de aşteptare se află mai multe documente spre a fi tipărite, prioritatea să îi fie acordată documentului cel mai vechi. Acest lucru nu numai că este corect, dar este şi necesar pentru a garanta că documentul nu aşteaptă la infinit. Astfel, pe sistemele mari este normal să se folosească cozi de tipărire.

Operaţiile fundamentale suportate de cozi sunt:

- *enqueue* - inserarea unui element la capătul cozii;
- *dequeue* - ştergerea primului element din coadă;
- *getFront* - accesul la primul element din coadă.

Figura 10.3 ilustrează operaţiile pe o coadă. Tradiţional, metodele `dequeue()` şi `getFront()` sunt combinate într-una singură. Prima metodă returnează primul element, după care îl scoate din coadă, în timp ce cea de-a doua returnează primul element fără a-l scoate din coadă.

Implementarea structurii de coadă este asemănătoare până la un punct cu cea a stivei. Coada păstrează elementele într-un şir de obiecte cu o capacitate iniţială, iar dacă prin adăugări repetate această capacitate este atinsă, atunci ea va fi mărită automat. Spre deosebire de stivă, coada are doi indici care indică poziţiile de început şi de sfârşit ale cozii.

Listing 10.7 ilustrează interfaţa pentru o coadă, în timp ce **Listing 10.8** prezintă o implementare a interfeţei anterioare, bazată pe şiruri. Spre deosebire de

stivă, în care adăugarea şi extragerea unui element au loc la acelaşi capăt, în cazul cozii adăugarea are loc la final, dar ştergerea se face de la început. Astfel, pentru a putea utiliza şirul `elements` la întreaga lui capacitate, elementele lui sunt privite "circular", ca şi când ultimul element ar fi legat de primul.

Listing 10.7: Interfaţă pentru coadă

```
package datastructures;

import exceptions.*;
/** Interfata pentru o coada. Expune metode pentru adaugarea
* unui element, stergerea primului element, consultarea
* primului element */
public interface Queue
{
  public void enqueue(Object x);

  public Object getFront() throws UnderflowException;

  public Object dequeue() throws UnderflowException;

  public boolean isEmpty();

  public void makeEmpty();
}
```

Listing 10.8: Implementarea unei structuri de tip coadă folosind şiruri

```
package datastructures;

import exceptions.*;
/** Implementarea unei cozi folosind un tablou. */
public class QueueAr implements Queue
{
  /** Tablou care retine elementele din coada*/
  private Object[] elements;
  /** Indicele primului element*/
  private int front;
  /** Indicele ultimului element*/
  private int back;
  /** Dimensiunea cozii */
  private int currentSize;
  /** Numarul de elemente alocat initial pentru coada*/
  private final static int DEFAULT_CAPACITY = 10;

  /**Constructor care aloca memorie pentru elementele cozii
  * si seteaza valorile atributelor*/
  public QueueAr()
  {
    elements = new Object[DEFAULT_CAPACITY];
```

```
    makeEmpty();
  }

  /** Intoarce true daca este vida.*/
  public boolean isEmpty()
  {
    return currentSize == 0;
  }

  /** Adauga elementul x in coada. */
  public void enqueue(Object x)
  {
    if (currentSize == elements.length)
    {
      increaseQueueSize();
    }

    back = increment(back);
    elements[back] = x;
    currentSize++;
  }

  /** Elimina toate elementele din coada. */
  public void makeEmpty()
  {
    currentSize = 0;
    front = 0;
    back = -1;
  }

  /** Extrage primul element din cadrul cozii.
  *@throws UnderflowException daca coada este goala*/
  public Object dequeue() throws UnderflowException
  {
    if (isEmpty())
    {
      throw new UnderflowException("Coada vida");
    }

    currentSize--;
    Object returnValue = elements[front];
    front = increment(front);

    return returnValue;
  }

  /** Intoarce primul element din cadrul cozii
  *@throws UnderflowException daca coada este goala.*/
  public Object getFront() throws UnderflowException
  {
```

```
    if (isEmpty())
    {
      throw new UnderflowException("Coada vida.");
    }

    return elements[front];
  }

  /**
  * Incrementeaza circular indicele din coada.*/
  */
  private int increment(int x)
  {
    if (++x == elements.length)
    {
      x = 0;
    }

    return x;
  }

  /** Incrementeaza dimensiunea cozii cu DEFAULT_CAPACITY atunci
  * cand coada este plina.*/
  private void increaseQueueSize()
  {
    Object[] newQueue = new Object[elements.length +
    DEFAULT_CAPACITY];

    for (int i = 0; i < elements.length; i++)
    {
      newQueue[i] = elements[i];
      front = increment(front);
    }

    elements = newQueue;
    front = 0;
    back = currentSize - 1;
  }
}
```

Exerciţiul 3 propune o a doua modalitate de a implementa această interfaţă, şi anume cu ajutorul listelor, care vor fi prezentate mai târziu în cadrul acestui capitol. **Listing 10.9** prezintă modul de utilizare a cozii. Deoarece operaţiile pe o coadă sunt restricţionate într-un mod asemănător cu operaţiile pe o stivă, este de aşteptat ca şi aceste operaţii să fie implementate într-un timp constant. Într-adevăr, toate operaţiile pe o coadă pot fi implementate în timp constant, *O(1)*.

Listing 10.9: Exemplu de utilizare a cozii. Programul va afişa: Conţinutul cozii este: 0 1 2 3 4

```
import datastructures.*;
import exceptions.*;
/** Clasa simpla de test pentru o coada. Se adauga 5 elemente,
* dupa care elementele sunt extrase pe rand */
public class TestQueue
{
  public static void main(String[] args)
  {
    Queue q = new QueueAr();

    //adaugam elemente in coada
    for (int i = 0; i < 5; i++)
    {
      q.enqueue(new Integer(i));
    }

    //scoatem si afisam elementele din coada
    System.out.print("Continutul cozii este: ");
    try
    {
      for (;;)
      {
        System.out.print(q.dequeue() + " ");
      }
    }
    catch(UnderflowException ue)
    {
    }
  }
}
```

10.4 Liste înlănţuite

Într-o listă înlănţuită elementele sunt reţinute discontinuu, spre deosebire de şiruri în care elementele sunt reţinute în locaţii continue. Acest lucru este realizat prin stocarea fiecărui obiect într-un *nod* care conţine obiectul şi o referinţă către următorul element în listă, ca în **Figura 10.4**. În acest model se reţin referinţe atât către primul (*first*) cât şi către ultimul (*last*) element din listă. Un nod al unei liste este implementat în **Listing 10.10.**

Listing 10.10: Un nod al unei liste

```java
package datastructures;
/**
 * Un nod al listei inlantuite. Clasa este vizibila
 * doar in pachet (package-friendly), deoarece este
 * pentru uzul intern al listei.
 */
class ListNode
{
  /** Valoarea continuta de nod.*/
  Object element;
  /** Legatura catre nodul urmator.*/
  ListNode next;

  public ListNode(Object element)
  {
    this.element = element;
    this.next = null;
  }

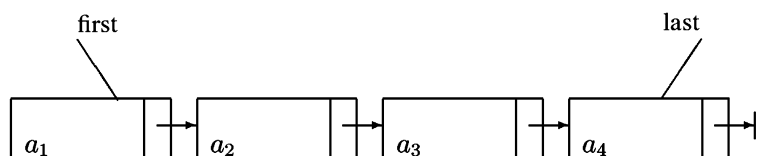
  public ListNode(Object element, ListNode next)
  {
    this.element = element;
    this.next = next;
  }
}
```

În orice moment, putem adăuga în listă un nou element x prin următoarele operaţii:

```java
last.next = new ListNode(x); //creaza un nod cu continutul x
last = last.next; //nodul creat anterior este ultimul in lista
```

În cazul unei liste înlănţuite un element oarecare nu mai poate fi găsit cu un singur acces. Aceasta este oarecum similar cu diferenţa între accesarea unei melodii pe CD (un singur acces) şi accesarea unei melodii pe casetă (acces secvenţial). Deşi din acest motiv listele pot să pară mai puţin atractive decât şirurile, există totuşi câteva avantaje importante. În primul rând, inserarea unui

Figura 10.4: O listă simplu înlănţuită.



element în mijlocul listei nu implică deplasarea tuturor elementelor de după punctul de inserare. Deplasarea datelor este foarte costisitoare (din punct de vedere al timpului), iar listele înlănţuite permit inserarea cu un număr constant de instrucţiuni de atribuire.

Merită observat că dacă permitem accesul doar la *first*, atunci obţinem o stivă, iar dacă permitem inserări doar la *last* şi accesări doar la *first*, obţinem o coadă. Exerciţiile 1 şi 3 de la finalul acestui capitol propun exact acest lucru: restricţionarea operaţiilor pe o listă pentru a obţine o stivă respectiv o coadă.

În general, atunci când folosim o listă, avem nevoie de mai multe operaţii, cum ar fi găsirea sau ştergerea unui element oarecare din listă. Trebuie să permitem şi inserarea unui nou element în orice punct. Aceasta este deja mult mai mult decât ne permite o stivă sau o coadă.

Pentru a accesa un element în listă, trebuie să obţinem o referinţă către nodul care îi corespunde. Evident că oferirea unei referinţe către un element încalcă principiul ascunderii informaţiei. Trebuie să ne asigurăm că orice acces la listă prin intermediul unei referinţe nu periclitează structura listei. Pentru a realiza acest lucru, lista este definită în două părţi: o clasă listă şi o clasă iterator. Clasa listă va reţine elementele propriu-zise ale listei în timp ce iteratorul va oferi o modalitate de a parcurge şi modifica elementele listei fără a periclita structura ei. **Listing 10.12** furnizează interfaţa de bază pentru o listă înlănţuită, oferind şi metodele care descriu doar starea listei.

Listing 10.14 defineşte o clasă iterator care este folosită pentru toate operaţiile de accesare a listei. Pentru a vedea cum funcţionează această clasă, să examinăm secvenţa de cod clasică pentru afişarea tuturor elementelor din cadrul unei structuri liniare. Dacă lista ar fi stocată într-un şir, secvenţa de cod ar arăta astfel:

```
//parcurge sirul a, afisand fiecare element
for (int index = 0; index < a.length; ++index)
{
    System.out.println(a[index]);
}
```

În Java elementar, codul pentru a itera o listă este:

```
//parcurge lista theList de tip List, afisand fiecare element
for (ListNode p = theList.first; p != null; p = p.next)
{
    System.out.println(p.data);
}
```

Pe de altă parte, trebuie avut în vedere faptul că anumite operaţii în cadrul listei pot eşua (de exemplu ştergerea unui element inexistent), motiv pentru care este utilizată următoarea clasa `ItemNotFoundException` din **Listing**

10.11.

Listing 10.11: Clasa de excepţii `ItemNotFoundException`

```
package exceptions;

/** Clasa de exceptii care semnaleaza tentativa de a cauta un
* element inexistent in cadrul unei structuri de date */
public class ItemNotFoundException extends Exception
{
  public ItemNotFoundException()
  {
    super();
  }

  public ItemNotFoundException(String msg)
  {
    super(msg);
  }
}
```

Listing 10.12: Interfaţă pentru o listă abstractă

```
package datastructures;

/** Interfata pentru un tip abstract de lista inlantuita. */
public interface List
{
  /** Testeaza daca lista e vida. */
  boolean isEmpty();

  /** Sterge toate elementele din lista. */
  void makeEmpty();
}
```

Listing 10.13: Implementarea unei liste înlănţuite

```
package datastructures;

/**
 * Lista simplu inlantuita.
 * Accesarea elementelor se face cu iteratorul LinkedListItr.
 */
public class LinkedList implements List
{
  ListNode header;


```

```
  public LinkedList()
  {
    header = new ListNode(null);
  }

  public boolean isEmpty()
  {
    return header.next == null;
  }

  public void makeEmpty()
  {
    header.next = null;
  }
}
```

Listing 10.14: Interfaţă pentru un iterator abstract de listă

```
package datastructures;

import exceptions.*;

/**
 * Interfata iterator pentru parcurgerea elementelor unei
 * liste inlantuite abstracte (simplu inlantuita, dublu
 * inlantuita, circulara, etc.)
 */
public interface ListItr
{
  /** Insereaza un element la pozitia curenta.*/
  void insert(Object x) throws ItemNotFoundException;

  /**
   * Seteaza pozitia curenta pe elementul x daca
   * il gaseste in lista.
   */
  boolean find(Object x);

  /** Sterge elementul x din lista. */
  void remove(Object x) throws ItemNotFoundException;

  /** Verifica daca lista a fost parcursa in totalitate. */
  boolean isInList();

  /** Obtine elementul aflat pe pozitia curenta. */
  Object retrieve();

  /** Seteaza pozitia inaintea primului element. */
  void zeroth();

```

```
  /** Seteaza pozitia curenta pe primul element. */
  void first();

  /** Avanseaza in lista la urmatorul element. */
  void advance();
}
```

Pornind de la interfaţa unei liste abstracte, codul pentru implementarea unei liste înlănţuite este dat în **Listing 10.13**, iar implementarea iteratorului unei liste înlănţuite este dată în **Listing 10.15**.

Mecanismul de iterare pe care l-ar folosi limbajul Java ar fi similar cu următoarea secvenţă:

```
//parcurge List, folosind abstractizarea si un iterator
ListItr itr = new LinkedListItr(theList);

for (itr.first(); itr.isInList(); itr.advance())
{
    System.out.println(itr.retrieve());
}
```

Iniţializarea dinaintea ciclului `for` creează un iterator al listei. Testul de terminare a ciclului foloseşte metoda `isInList()` definită pentru clasa `LinkedListItr`. Metoda `advance()` trece la următorul nod din cadrul listei. Putem accesa elementul curent prin apelul metodei `retrieve()` definită în `LinkedListItr`. Principiul general este că accesul fiind realizat prin intermediul clasei `ListItr`, securitatea datelor este garantată. Putem avea mai mulţi iteratori care să traverseze simultan o singură listă.

Pentru a funcţiona corect, clasa `ListItr` trebuie să menţină două obiecte. În primul rând, are nevoie de o referinţă către nodul curent. În al doilea rând are nevoie de o referinţă către obiectul de tip `List` pe care îl indică (această referinţă este iniţializată o singură dată în cadrul constructorului).

Listing 10.15: Implementarea iteratorului listei înlănţuite

```
package datastructures;

import exceptions.*;

/**
* Iterator pentru lista inlantuita.
*/
public class LinkedListItr implements ListItr
{
  /** Referinta catre lista care va fi iterata */
  protected LinkedList theList;
  /**Referinta catre nodul curent.*/
  protected ListNode current;
```

```

/**Construieste un iterator pe lista list*/
public LinkedListItr(LinkedList list)
{
  theList = list;
  current = list.isEmpty() ? list.header : list.header.next;
}

/** Construieste cu parametru de tip List. Daca list
* nu refera o instanta LinkedList, se arunca o exceptie
* de tipul ClassCastException */
public LinkedListItr(List list) throws ClassCastException
{
  this((LinkedList) list);
}

/** Reseteaza iteratorul care va indica pozitia dinaintea
* primului element din lista*/
public void zeroth()
{
  current = theList.header;
}

/** Adauga obiectul x la pozitia curenta in lista
* @throws ItemNotFoundException daca lista este vida */
public void insert(Object x) throws ItemNotFoundException
{
  if (current == null)
  {
    throw new ItemNotFoundException("Eroare la inserare");
  }

  ListNode newNode = new ListNode(x, current.next);
  current.next = newNode;
  current = current.next;
}
/** Intoarce true daca obiectul x se afla in lista */
public boolean find(Object x)
{
  ListNode itr = theList.header.next;

  while (itr != null && !itr.element.equals(x))
  {
    itr = itr.next;
  }

  if (itr == null)
  {
    return false;
  }
```

```

  current = itr;
  return true;
}

/** Sterge obiectul x din lista
* @throws ItemNotFoundException daca x nu se afla in lista*/
public void remove(Object x) throws ItemNotFoundException
{
  ListNode itr = theList.header;

  while (itr.next != null && !itr.next.element.equals(x))
  {
    itr = itr.next;
  }

  if (itr.next == null)
  {
    throw new ItemNotFoundException("Stergere esuata");
  }

  itr.next = itr.next.next; //trecem peste nodul sters
  current = theList.header;
}

/** Intoarce elementul de pe pozitia curenta */
public Object retrieve()
{
  return (isInList() ? current.element : null);
}

/** Intoarce true daca ne aflam in interiorul listei */
public boolean isInList()
{
  return (current != null) && (current != theList.header);
}

/** Elementul curent va fi primul element din lista */
public void first()
{
  current = theList.header.next;
}
/** Avanseaza iteratorul pe urmatorul element*/
public void advance()
{
  if (current != null)
  {
    current = current.next;
  }
}
```

```
}
```

Iată şi exemplul de utilizare a listei:

Listing 10.16: Exemplu de utilizare a listei. Programul va afişa: Conţinutul listei: 4 3 2 1 0

```
import datastructures.*;
import exceptions.*;
/** Clasa simpla pentru testarea unei liste, care adauga 5 numere,
* dupa care afiseaza continutul listei.*/
public class TestList
{
  public static void main(String[] args)
  {
    List theList = new LinkedList();
    ListItr itr = new LinkedListItr(theList);

    //se insereaza elemente pe prima pozitie
    for (int i = 0; i < 5; i++)
    {
      try
      {
        itr.insert(new Integer(i));
      }
      catch(ItemNotFoundException e)
      {
      }

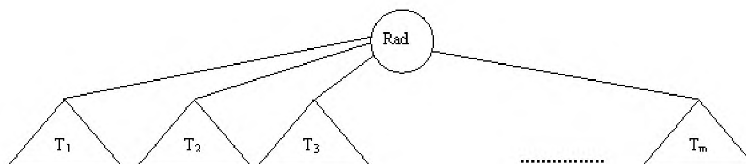
      itr.zeroth(); //se trece la inceputul listei
    }

    System.out.print("Continutul listei: ");

    for (itr.first(); itr.isInList(); itr.advance())
    {
      System.out.print(itr.retrieve() + " ");
    }
  }
}
```

Deşi discuţia s-a axat pe liste simplu înlănţuite, interfeţele din **Listing 10.12** şi **Listing 10.14** pot fi folosite pentru oricare tip de listă, indiferent de implementarea pe care o are la bază. Interfaţa nu precizează faptul că este nevoie de liste simplu înlănţuite.

Figura 10.5: Un arbore generic



10.5 Arbori

10.5.1 Noţiuni generale

Vom trece acum să studiem cele mai importante structuri neliniare care apar în algoritmii pentru calculatoare: arborii. În general vorbind, structura arborescentă implică o relaţie de ramificare între noduri, foarte asemănătoare celei întâlnite la crengile unui arbore din natură.

Una dintre definiţiile cele mai răspândite ale arborilor (nu neapărat binari) este următoarea (după D.E. Knuth):

Definiţie: Un arbore (**Figura 10.5**) este o mulţime finită T de unul sau mai multe noduri, care are proprietăţile:

i) există un nod special, numit rădăcina arborelui;

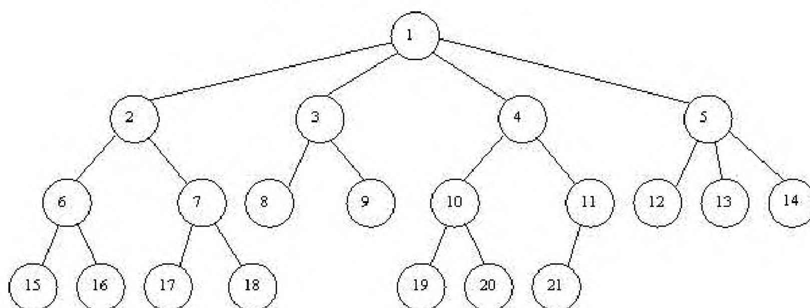
ii) toate celelalte noduri din T sunt repartizate în mulţimi $T_1, T_2, \ldots, T_m$ disjuncte, fiecare mulţime la rândul său fiind un arbore. Arborii $T_1, T_2, \ldots, T_m$ se numesc subarborii rădăcinii.

Se observă că definiţia de mai sus este recursivă (recursivitatea este prezentată în capitolul 12): am definit un arbore pe baza unor arbori. Totuşi, privind cu atenţie definiţia ne dăm seama că nu se pune problema circularităţii, deoarece un arbore cu un singur nod este alcătuit doar din rădăcină, iar arborii cu $n > 1$ noduri sunt definiţi pe baza arborilor cu mai puţin de n noduri. Există şi definiţii nerecursive ale arborilor[3], dar definiţia recursivă este mai adecvată, deoarece vom vedea că recursivitatea pare să fie o trăsătură inerentă operaţiilor pe structuri arborescente. Caracterul recursiv al arborilor este de altfel prezent şi în natură, deoarece mugurii arborilor tineri cresc şi se dezvoltă în subarbori, care la rândul lor fac muguri şi aşa mai departe.

Nodul rădăcină al fiecărui subarbore se numeşte fiul (sau copilul) rădăcinii, iar rădăcina este tatăl (sau părintele) fiecărui nod rădăcină din subarbori.

[3]De exemplu, în teoria grafurilor, un arbore este definit ca un graf conex şi fără cicluri.

Figura 10.6: Un arbore oarecare.



Din definiţia recursivă reiese că un arbore este o colecţie de n noduri, dintre care unul este rădăcina, şi $n-1$ muchii. Faptul că există $n-1$ muchii se deduce din observaţia simplă că fiecare muchie leagă un nod de părintele său şi fiecare nod, în afară de rădăcină, are exact un părinte.

În arborele din **Figura 10.6**, rădăcina este `1`. Nodul `5` îl are ca părinte pe `1` şi are fii `12,13` şi `14`. Nodurile care nu au fii se numesc frunze. Frunzele din arborele de mai sus sunt `8,9,12,13,...,21`. Nodurile cu acelaşi părinte se numesc fraţi. În mod asemănător se definesc relaţiile nepot, bunic etc.

Un drum de la un nod n_1 la un nod n_k este o secvenţă de noduri $n_1, n_2, \ldots, n_k$ astfel încât n_i este tatăl lui n_{i+1} pentru $i = 1, k-1$. Lungimea unui drum este dată de numărul de muchii ale drumului, adică $k-1$. Pentru oricare nod n_i, nivelul (adâncimea) nodului este lungimea unicului drum de la rădăcină la n_i. Astfel, nivelul rădăcinii este 0. În arborele din **Figura 10.6**, nivelul nodului `2` este 1, iar al nodului `15` este 3. Adâncimea unui arbore este definită ca fiind maximul adâncimilor nodurilor. În exemplul nostru, adâncimea arborelui este egală cu adâncimea nodului `15,` care este 3.

10.5.2 Arbori binari

Definiţie: Un arbore binar este un arbore în care orice nod are cel mult doi fii.

Figura 10.7 arată că un arbore binar constă într-o rădăcină şi doi subarbori T_s şi T_d, oricare din ei putând să fie vid (să lipsească).

O proprietate deosebit de importantă a arborilor binari este că adâncimea medie a unui arbore binar cu n noduri este considerabil mai mică decât n. Se poate arăta că adâncimea medie a unui arbore binar este proporţională cu $\sqrt{n}$, iar adâncimea medie a unui caz particular de arbore binar, arborele binar de

Figura 10.7: Arbore binar generic

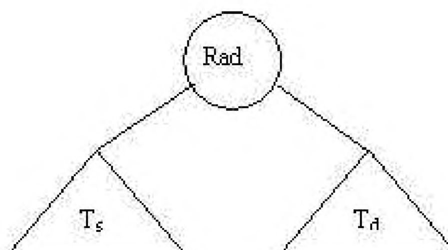
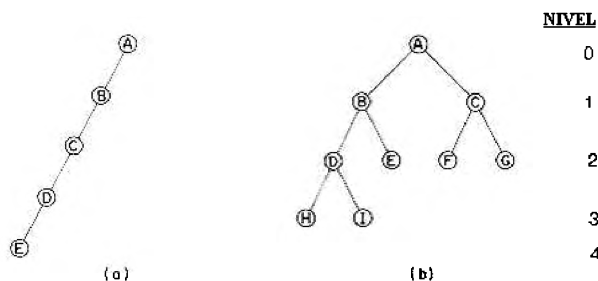


Figura 10.8: Exemple de arbori binari (a) arbore binar degenerat (b) arbore binar nedegenerat



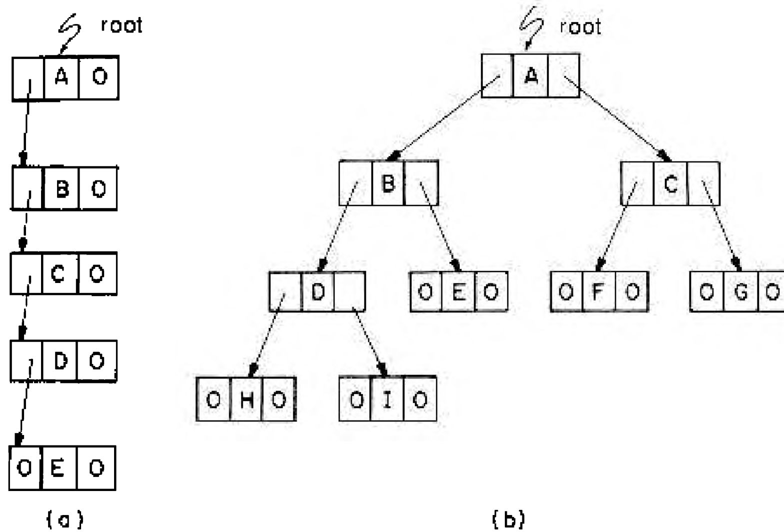
căutare (secţiunea 10.5.3), este proporţională cu $\log n$. Din păcate, în cazurile degenerate, adâncimea unui arbore poate fi chiar $n - 1$, după cum se vede şi în **Figura 10.8(a)**.

Înălţimea (adâncimea) unui arbore binar este foarte importantă, deoarece multe operaţii definite asupra arborilor (ştergere de nod, inserare de nod etc.) sunt proporţionale cu înălţimea arborelui. Din acest motiv, s-a recurs la diferite metode pentru a menţine adâncimea unui arbore proporţională cu $\log n$ în orice situaţie (arbori AVL, arbori bicolori etc).

Parcurgerea arborilor binari

După cum reiese şi din **Figura 10.7**, putem defini un arbore binar ca fiind o mulţime finită de noduri care este fie vidă, fie este formată dintr-o rădăcină şi doi arbori binari. Această definiţie sugerează o metodă naturală de reprezentare a arborilor binari: fiecare nod va fi descris de trei componente: `element` - informaţia utilă a nodului, `left` - o referinţă către fiul din stânga şi `right` -

Figura 10.9: Reprezentarea înlănţuită a arborilor din **Figura** 10.8



o referinţă către fiul din dreapta. Dacă un nod nu are fiu în stânga atunci `left` este `null`, analog, dacă nu are fiu în dreapta, atunci `right` este `null`. Putem astfel defini clasa `BinaryNode`, care reţine un nod al unui arbore binar:

```
public class BinaryNode
{
  Object element ;
  BinaryNode left ;
  BinaryNode right ;
   ...
   //constructori si alte metode
}
```

Informaţia utilă din cadrul nodului este reţinută de atributul `element`, care este de tip `Object`, deci poate referi o instanţă de orice tip. **Figura 10.9** prezintă modul în care se reprezintă înlănţuit arborii binari din **Figura 10.8**. Variabila `root` din **Figura 10.9** este o referinţă care indică rădăcina arborelui.

Există mai mulţi algoritmi pentru manevrarea structurilor arborescente, şi o idee care apare de foarte multe ori este noţiunea de parcurgere, de "deplasare" prin arbore. Parcurgerea unui arbore este de fapt o metodă de examinare sistematică a nodurilor arborelui în aşa fel încât fiecare nod să fie vizitat o singură

dată. Parcurgerea completă a unui arbore ne oferă o aranjare lineară a nodurilor, şi operarea multor algoritmi este simplificată dacă ştim care este următorul nod la care ne vom deplasa într-o astfel de secvenţă, pornind de la un nod dat.

Există trei moduri principale în care un arbore binar poate fi parcurs: nodurile se pot vizita în preordine, inordine şi în postordine. Vom descrie aceste trei metode recursiv, ca şi definiţia arborelui.

Dacă un arbore binar este vid (nu are nici un nod) parcurgerea lui nu presupune nici o operaţie; în caz contrar parcurgerea comportă trei etape, descrise în tabelul următor:

Preordine	**Inordine**	**Postordine**
Se vizitează rădăcina	Se parcurge subarborele stâng	Se parcurge subarborele stâng
Se parcurge subarborele stâng	Se vizitează rădăcina	Se parcurge subarborele drept
Se parcurge subarborele drept	Se parcurge subarborele drept	Se vizitează rădăcina

Pentru arborele din **Figura 10.8** găsim că nodurile în preordine sunt:

`A B D H I E C F G`

deoarece mai întâi se vizitează rădăcina `A`, apoi subarborele stâng (`B D H I E`) şi apoi subarborele drept (`C F G`).

Similar, parcurgerea în inordine are ca rezultat şirul:

`H D I B E A F C G`

iar parcurgerea în postordine:

`H I D E B F G C A`

Numele de preordine, inordine şi postordine derivă, bineînţeles de la poziţia relativă a rădăcinii faţă de subarbori. O posibilă metodă de a parcurge un arbore în preordine este dată în **Listing 10.17**. Metoda primeşte ca parametru o referinţă către rădăcina subarborelui care este vizitat. Evident că la primul apel, parametrul este chiar rădăcina arborelui (referinţa `root` din **Figura 10.9**). Vom vedea cum se integrează parcurgerea unui arbore în cadrul unei clase complete în paragraful 10.5.3.

Listing 10.17: Metodă generică pentru parcurgerea în preordine a unui arbore binar

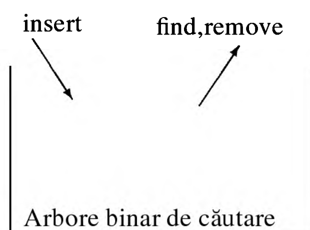
```
public void preord(BinaryNode t)
{
  if( t != null )
  {
    process( t.element ) ;
    preord( t.left ) ;
    preord( t.right ) ;
  }
}
```

Celelalte două parcurgeri se realizează absolut analog, prin simpla reordonare a instrucţiunilor de parcurgere.

10.5.3 Arbori binari de căutare

Una dintre cele mai importante aplicaţii ale arborilor este utilizarea acestora în probleme de căutare. Proprietatea care face ca un arbore binar să devină un arbore binar de căutare este: pentru oricare nod X al arborelui toate nodurile din subarborele stâng sunt mai mici decât X, şi toate nodurile din subarborele drept sunt mai mari decât X. Aşadar, faţă de arborii binari obişnuiţi, arborii de căutare mai adaugă o relaţie de ordine între elemente. Pentru a putea fi comparate, nodurile nu var mai conţine obiecte de tip `Object`, ca în paragraful 10.5.2, ci instanţe ale clasei `Comparable`. Arborele binar de căutare este o structură de date în care, pe lângă căutare rapidă, putem adăuga sau şterge eficient elemente. **Figura 10.10** ilustrează operaţiile de bază permise asupra unui arbore binar de căutare.

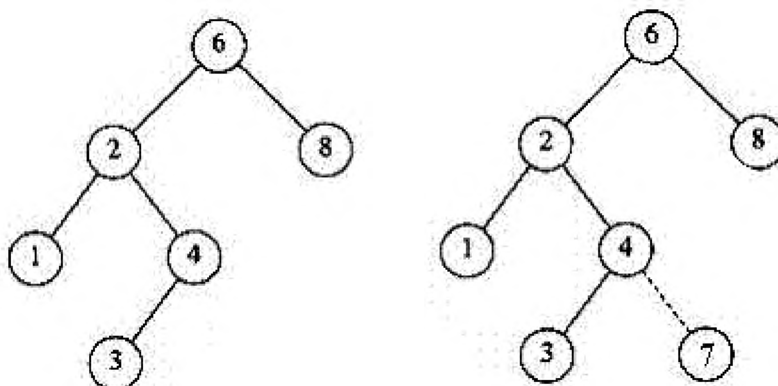
Figura 10.10: Modelul pentru arborele binar de căutare.



De exemplu, arborii din **Figura 10.8** nu sunt arbori binari de căutare, deoarece, în ambele cazuri, în stânga nodului `A` se află nodul `B`, care are valoare mai mare decât `A` (dacă considerăm ordinea alfabetică).

Să studiem cu atenţie arborii din **Figura 10.11**. La prima vedere, ambii par să fie arbori binari de căutare; examinând totuşi mai minuţios arborele din dreapta, constatăm că nodul 7, deşi este mai mare decât rădăcina, 6, se află în stânga ei, ceea ce contravine regulii arborilor de căutare.

Figura 10.11: Doi arbori binari. Doar arborele din stânga este şi arbore binar de căutare.



Principalele operaţii care se realizează asupra arborilor binari de căutare sunt:

- Căutarea unui nod în arbore;
- Găsirea minimului şi maximului;
- Inserarea unui nou nod;
- Ştergerea unui nou nod;
- Vidarea arborelui (ştergerea tuturor nodurilor).

O posibilă interfaţă care descrie aceste operaţii este prezentată în **Listing 10.19**.

Setul de operaţii permise este acum extins faţă de stivă şi coadă pentru a permite găsirea unui element arbitrar, alături de inserare şi ştergere. Metoda `find()` întoarce o referinţă către un obiect care este egal (în sensul metodei `compareTo()` din interfaţa `Comparable`) cu elementul căutat. Dacă nu există nici un element egal cu cel căutat, `find()` aruncă o excepţie `ItemNotFoundException`, prezentată în secţiunea anterioară. Aceasta este o decizie de proiectare. O altă posibilitate ar fi fost să întoarcem `null` în cazul în care elementul căutat nu este găsit. Diferenţa între cele două abordări constă în faptul că metoda noastră îl obligă pe programator să trateze explicit situaţia în care căutarea nu are succes. În cealaltă situaţie, dacă am fi întors `null`, şi nu s-ar fi făcut verificările necesare, programul ar fi generat un

Listing 10.18: Clasa de excepţie `DuplicateItemException`

```
package exceptions;

/** Clasa de exceptii care semnaleaza adaugarea unui element deja
* existent intr-o structura de date */
public class DuplicateItemException extends Exception
{
  public DuplicateItemException()
  {
    super();
  }

  public DuplicateItemException(String msg)
  {
    super(msg);
  }
}
```

`NullPointerException` la prima tentativă de a folosi referinţa. Din punct de vedere al eficienţei, versiunea cu excepţii ar putea fi ceva mai lentă, dar este puţin probabil ca rezultatul să fie observabil, cu excepţia situaţiei în care codul ar fi foarte des executat în cadrul unor situaţii în care viteza este critică.

În mod similar, inserarea unui element care deja este în arbore este semnalată printr-o excepţie `DuplicateItemException` (vezi **Listing 10.18**). Există şi alte posibile alternative. Una dintre ele constă în a permite noii valori să suprascrie valoarea stocată.

Vom da ca exemplu un arbore de căutare care reţine stringuri. Acest lucru este posibil deoarece `String` implementează interfaţa `Comparable`. Implementarea arborelui binar în conformitate cu interfaţa `SearchTree` este prezentată în **Listing 10.21**. Pentru a implementa nodurile arborelui binar se utilizează clasa `BinaryNode`, prezentată în **Listing 10.20**.

Implementarea arborelui binar de căutare este mult mai complexă decât pentru stivă sau coadă şi foloseşte recursivitatea (care va fi prezentată în capitolul 12). Nu vă îngrijoraţi dacă nu înţelegeţi acum toate detaliile de implementare; puteţi trece mai departe fără nici o problemă şi reveni mai târziu, după ce aţi parcurs şi înţeles capitolul 12.

Toate elementele din arborele binar ne sunt accesibile prin intermediul rădăcinii sale, care este reţinută în atributul `root`. Să descriem acum pe scurt modul de funcţionare al metodelor din clasa `BinarySearchTree`.

Listing 10.19: Interfaţa pentru un arbore binar de căutare

```
package datastructures;

import exceptions.*;

/**
* Interfata pentru arbori binari de cautare. Expune metode
* pentru adaugarea si stergerea unui element oarecare si
* pentru adaugarea si stergerea elementului minim si maxim.
*/
public interface SearchTree
{
  void insert(Comparable x) throws DuplicateItemException;
  void remove(Comparable x) throws ItemNotFoundException;
  void removeMin() throws ItemNotFoundException;
  Comparable find(Comparable x) throws ItemNotFoundException;
  Comparable findMin() throws ItemNotFoundException;
  Comparable findMax() throws ItemNotFoundException;
  boolean isEmpty();
  void makeEmpty();
}
```

Căutarea unui nod în arbore: Aşa cum le spune şi numele, arborii binari de căutare au fost concepuţi pentru a se putea căuta informaţiile rapid şi uşor. Să presupunem că în arborele de căutare din **Figura 10.11** (cel din stânga), dorim să găsim nodul care are cheia 4. Pentru aceasta comparăm valoarea 4 cu rădăcina. Deoarece 4<6, nodul căutat se află în subarborele stâng, care are rădăcina 2. Comparăm 4 cu 2. Deoarece 4>2, nodul căutat se află în subarborele drept. Comparăm pe 4 cu fiul din dreapta al lui 2, constatăm că valorile sunt egale şi ne oprim. Dacă în loc de cheia 4 am fi căutat cheia 5, atunci am fi ajuns în subarborele drept al lui 4, care este vid, deci nodul 5 nu se găseşte în arbore.

Căutarea unui nod se realizează cu metoda `find()`, implementată în liniile 75-95 din **Listing 10.21**. Această metodă porneşte cu căutarea de la rădăcina arborelui şi coboară fie către stânga fie către dreapta (funcţie de relaţia dintre elementul căutat şi rădăcina subarborelui) până când elementul căutat este găsit sau s-a "ieşit" din arbore, caz în care se aruncă o `ItemNotFoundException()`.

Găsirea elementului minim şi a celui maxim: Găsirea elementului minim (găsirea maximului se face analog) constă pur şi simplu în a coborî în arbore pe partea stângă (liniile 105-108) până se ajunge la frunză. Elementul din frunză este chiar minimul.

Listing 10.20: Clasa care modelează un nod al unui arbore binar

```
package datastructures;

/**
* Reprezinta un nod in arbore. Este pentru uz intern.
*/
class BinaryNode
{
  Comparable element;
  BinaryNode left;
  BinaryNode right;

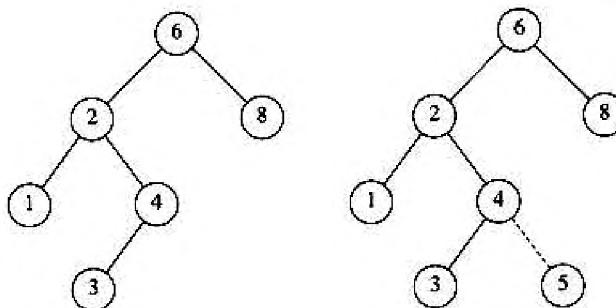

  BinaryNode(Comparable e)
  {
    element = e;
    left = null;
    right = null;
  }

  BinaryNode(Comparable e, BinaryNode lt, BinaryNode rt)
  {
    element = e;
    left = lt;
    right = rt;
  }
}
```

Adăugarea unui nod în arbore: Adăugarea unui nod în arbore se realizează folosind metoa `insert(Comparable)` din liniile 20-24, care la rândul ei delegă problema către `insert(Comparable,BinaryNode)` de la liniile 129-150. Probabil că vă întrebaţi de ce a fost nevoie să scindăm operaţia de adăugare în două metode? Metoda propriu-zisă de ştergere are şi un parametru de tip `BinaryNode`, care la început este chiar `root`, rădăcina arborelui. Utilizatorul obişnuit nu are acces la acest atribut, de aceea el poate apela doar metoda `insert(Comparable)`, care fiind o metodă din cadrul clasei, va putea apela la linia 23 `insert(x,root)`. Această tehnică este folosită pentru toate metodele existente în clasa `BinarySearchTree`.

Metoda de adăugare a unui nod în arbore este conceptual destul de simplă. Pentru a insera `x` în arborele cu rădăcina `t`, ne “afundăm” în interiorul arborelui exact ca şi în cazul metodei `find()`. Dacă valoarea `x` este găsită în arbore, atunci aruncăm o `DuplicateItemException`. În caz contrar inserăm un nod cu cheia `x`, în ultimul punct din calea care a fost parcursă. **Figura 10.12**

Figura 10.12: Arborele binar de căutare înainte şi după inserarea nodului 5.



ne arată ce se petrece. Pentru a insera valoarea 5, traversăm arborele ca şi când am realiza procesul de căutare. Când ajungem la nodul cu cheia 4, trebuie să mergem către dreapta, dar nu există subarbore drept, deci 5 nu este în arbore şi acesta este locul în care trebuie inserat.

Ca orice metodă recursivă, `insert()` începe la linia 132 cu condiţia de terminare:

```
if( t == null )
```

care este adevărată în momentul în care subarborele curent este vid. În acest caz se creează pur şi simplu un nou nod în care se plasează obiectul x şi se actualizeză rădăcina subarborelui cu nodul curent. În cazul în care subarborele în care se inserează x nu este vid, comparăm x cu valoarea din rădăcina arborelui:

```
x.compareTo( t.element )
```

Dacă valoarea din x este mai mică, atunci x se va insera în subarborele stâng:

```
t.left = insert(x, t.left);
```

altfel, dacă este mai mare se va insera în subarborele drept:

```
t.right = insert(x, t.right);
```

iar dacă valorile sunt egale înseamnă că nodul respectiv exista deja în arbore şi se aruncă o `DuplicateItemException`.

Metoda `insert()` conţine un aspect de fineţe: cum se leagă noul nod inserat (5 în exemplul nostru) de tatăl său (4)? În metoda `insert()` nu pare să existe o secvenţă de cod care să semnaleze că fiul din dreapta al lui 4 nu mai este `null`, ci este noul nod inserat. Răspunsul se află în modul în care se

realizează apelul recursiv al metodei `insert()`: subarborelui stâng (sau drept) curent i se *atribuie* rezultatul inserării nodului x curent. La un moment dat, metoda va crea un nou nod, iar adresa acelui nod va fi întoarsă către nodul părinte.

Ştergerea elementului minim şi a celui maxim: Ştergerea elementului minim constă în găsirea lui, după care se "ridică" subarborele drept în locul lui. Metoda `removeMin()` de la liniile 152-170 realizează exact acest lucru. Observaţi că aici, pentru variaţie, ciclul `while` de la liniile 105-108 ale metodei `findMin()` a fost înlocuit cu un apel recursiv. "Ridicarea" subarborelui drept în locul elementului şters se face simplu, prin atribuirea de la linia 166:

```
t = t.right ;
```

Ştergerea elementului maxim se realizează absolut analog.

Ştergerea unui nod din arbore: Operaţia de ştergere a unui nod este cea mai delicată operaţie pe arborele binar de căutare, deoarece pe lângă eliminarea nodului mai implică şi operaţii suplimentare pentru a păstra structura de arbore binar. În cazul operaţiei de ştergere, după ce am găsit nodul care trebuie şters trebuie să considerăm mai multe posibilităţi:

- Dacă nodul care trebuie şters este o frunză, el poate fi şters imediat prin înlocuirea legăturii părintelui său cu `null`;
- Dacă nodul care trebuie şters are doar un singur fiu (indiferent că este stâng sau drept), ştergerea lui se face prin ajustarea legăturii părintelui său pentru a-l "ocoli" (**Figura 10.13**);
- Cazul complicat apare atunci când nodul care trebuie şters are doi fii. Strategia generală în acest caz este de a înlocui cheia nodului care trebuie şters cu cea mai mică cheie din subarborele său drept (care este uşor de găsit) după care nodul cu această cheie se şterge folosind metoda `removeMin()` descrisă anterior (**Figura 10.14**).

Ştergerea unui nod din arbore se realizează folosind metoda `remove(Comparable)` din liniile 28-32, care la rândul ei delegă problema către `remove(Comparable,BinaryNode)` de la liniile 172-199. Ca orice metodă recursivă, `remove()` începe la linia 175 cu condiţia de terminare:

```
if( t == null )
```

Figura 10.13: Ştergerea unui nod (4) cu un singur fiu. Legătura 2-3 "ocoleşte" nodul 4.

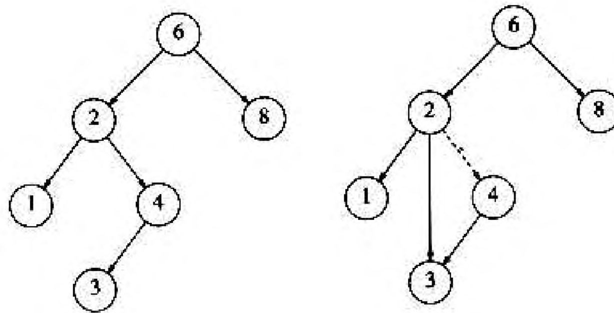
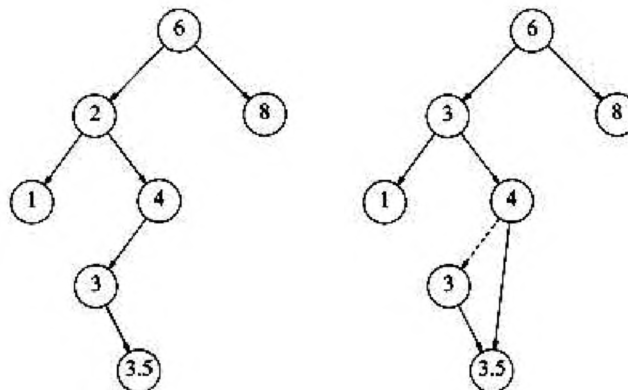


Figura 10.14: Ştergerea unui nod cu doi fii. Cel mai mic nod din subarborele drept al lui 2 este 3. Se înlocuieşe 2 cu 3, după care 3 se şterge prin "ocolire"



care este adevărată când subarborele curent este vid. În această situaţie elementul `x` nu este găsit şi se aruncă o `ItemNotFoundException`.

Dacă subarborele nu este vid, se compară elementul care trebuie şters cu rădăcina şi, dacă sunt diferite, se coboară fie pe subarborele drept fie pe cel stâng. În cazul în care elementul care trebuie şters a fost găsit (adică `x.compareTo(t.element)` este 0) trebuie să eliminăm nodul respectiv din arbore având însă mare grijă să nu pierdem structura de arbore binar de căutare. Dacă nodul curent are descendenţi atât în stânga cât şi în dreapta, adică

```
t.left != null && t.right != null
```

este `true`, vom folosi artificiul descris anterior: vom aduce în locul nodului şters, cel mai mic nod din subarborele drept (care este evident mai mare decât toate nodurile din subarborele stâng, deci se va păstra structura de arbore de căutare):

```
t.element = findMin(t.right).element ;
```

după care vom apela `removeMin()` pentru a şterge elementul minim din acest subarbore:

```
t.right = removeMin(t.right)
```

Dacă nodul curent nu are descendenţi în ambele părţi, operaţia de ştergere este mult mai simplă: pur şi simplu se "urcă" subarborele nevid în locul nodului care a fost şters (altfel spus, nodul şters este "ocolit"):

```
t = (t.left != null)? t.left : t.right ;
```

Metodele `isEmpty()` şi `makeEmpty()` sunt banale şi nu le vom detalia aici.

Listing 10.21: Implementarea arborelui binar de căutare

```
package datastructures;

import exceptions.*;
/** Arbore binar de cautare alocat inlantuit. Toate
* cautarile se fac pe baza metodei compareTo(). Expune
* metode pentru inserare, stergere, cautare etc. */
public class BinarySearchTree implements SearchTree
{
  /** Referinta catre radacina arborelui*/
  protected BinaryNode root;

  /**Constructor care initializeaza radacina cu null */
  public BinarySearchTree()
```

```
{
  root = null;
}

/** Adauga elementul x in arbore
* @throws DuplicateItemException daca elementul deja exista.*/
public void insert(Comparable x)
throws DuplicateItemException
{
  root = insert(x, root);
}

/** Sterge elementul x din arbore.
* @throws ItemNotFoundException daca elementul nu se afla
*           in arbore.*/
public void remove(Comparable x)
throws ItemNotFoundException
{
  root = remove(x, root);
}

/** Sterge elementul minim din arbore
*  @throws ItemNotFoundException daca arborele este vid*/
public void removeMin() throws ItemNotFoundException
{
  root = removeMin(root);
}

/** Intoarce elementul minim din arbore.
*  @throws ItemNotFoundException daca arborele este vid*/
public Comparable findMin() throws ItemNotFoundException
{
  return findMin(root).element;
}

/** Intoarce elementul maxim din arbore.
*  @throws ItemNotFoundException daca arborele este vid*/
public Comparable findMax() throws ItemNotFoundException
{
  return findMax(root).element;
}

/** Cauta elementul x in arbore.
*  @throws ItemNotFoundException daca x nu este gasit.*/
public Comparable find(Comparable x)
throws ItemNotFoundException
{
  return find(x, root).element;
}

```

```java
/** Intoarce true daca arborele este vid */
public boolean isEmpty()
{
  return root == null;
}

/**Videaza arborele, setand radacina la null */
public void makeEmpty()
{
  root = null;
}

protected BinaryNode find(Comparable x, BinaryNode t)
throws ItemNotFoundException
{
  while (t != null)
  {
    if (x.compareTo(t.element) < 0)
    {
      t = t.left;
    }
    else if (x.compareTo(t.element) > 0)
    {
      t = t.right;
    }
    else
    {
      return t; //element gasit
    }
  }

  throw new ItemNotFoundException("Element negasit.");
}

protected BinaryNode findMin(BinaryNode t)
throws ItemNotFoundException
{
  if (t == null)
  {
    throw new ItemNotFoundException("Element negasit");
  }

  while (t.left != null)
  {
    t = t.left;
  }

  return t;
}

```

```
protected BinaryNode findMax(BinaryNode t)
throws ItemNotFoundException
{
  if (t == null)
  {
    throw new ItemNotFoundException("Element negasit");
  }

  while (t.right != null)
  {
    t = t.right;
  }

  return t;
}

protected BinaryNode insert(Comparable x, BinaryNode t)
throws DuplicateItemException
{
  if (t == null)
  {
    t = new BinaryNode(x, null, null);
  }
  else if (x.compareTo(t.element) < 0)
  {
    t.left = insert(x, t.left);
  }
  else if (x.compareTo(t.element) > 0)
  {
    t.right = insert(x, t.right);
  }
  else
  {
    throw new DuplicateItemException("Elementul exista deja");
  }

  return t;
}

protected BinaryNode removeMin(BinaryNode t)
throws ItemNotFoundException
{
  if (t == null)
  {
    throw new ItemNotFoundException("Element negasit");
  }

  if (t.left != null)
  {
    t.left = removeMin(t.left);
```

```
    }
    else
    {
      t = t.right;
    }

    return t;
  }

  protected BinaryNode remove(Comparable x, BinaryNode t)
  throws ItemNotFoundException
  {
    if (t == null)
    {
      throw new ItemNotFoundException("Element negasit");
    }

    if (x.compareTo(t.element) < 0)
    {
      t.left = remove(x, t.left);
    }
    else if (x.compareTo(t.element) > 0)
    {
      t.right = remove(x, t.right);
    }
    else if (t.left != null && t.right != null)
    {
      t.element = findMin(t.right).element;
      t.right = removeMin(t.right);
    }
    else
    {
      t = (t.left != null) ? t.left : t.right;
    }

    return t;
  }

}
```

Listing 10.22 prezintă modul în care arborele binar de căutare poate fi folosit pentru obiecte de tip `String`.

Interfaţa `SearchTree` mai are două metode suplimentare: una pentru a găsi cel mai mic element şi una pentru a găsi cel mai mare element. Se poate arăta că transpirând un pic mai mult se poate găsi foarte eficient şi cel mai mic al *k*-lea element, pentru oricare *k* trimis ca parametru.

Să vedem care sunt timpii de execuţie pentru operaţiile pe un arbore binar de căutare. Este normal să sperăm că timpii de execuţie pentru `find()`,

`insert()` şi `remove()` să fie logaritmici, deoarece aceasta este valoarea pe care o vom obţine pentru căutarea binară (paragraful 12.3). Din nefericire, pentru cea mai simplă implementare a arborelui binar de căutare, acest lucru nu este adevărat. Timpul mediu de execuţie este într-adevăr logaritmic, dar în cazul cel mai nefavorabil (când arborele este "dezechilibrat") timpul de execuţie este `O(n)`, caz care apare destul de frecvent. Totuşi, prin aplicarea anumitor trucuri de algoritmică se pot obţine anumite structuri mai complexe (arbori bicolori) care au într-adevăr un cost logaritmic pentru fiecare operaţie.

Listing 10.22: Model de program care utilizează arbori de căutare. Programul va afişa: Găsit Cristi;Alina nu a fost găsită

```
import datastructures.*;
import exceptions.*;
/** Clasa simpla de test pentru arborii binari de cautare.*/
public class TestSearchTree
{
  public static void main(String[] args)
  {
    SearchTree t = new BinarySearchTree();

    String result = null;

    try
    {
      t.insert("Cristi");
    }
    catch(DuplicateItemException e)
    {
    }

    try
    {
      result = (String) t.find("Cristi");

      System.out.print("Gasit " + result + "; ");
    }
    catch(ItemNotFoundException e)
    {
      System.out.print("Cristi nu a fost gasit; ");
    }

    try
    {
      result = (String) t.find("Alina");

      System.out.print("Gasit " + result + " ;");
    }
    catch(ItemNotFoundException e)
```

```
        {
            System.out.print("Alina nu a fost gasita;");
        }
    }
}
```

Ce putem spune despre operaţiile `findMin()` şi `findMax()`? În mod cert, aceste operaţii necesită un timp constant în cazul căutării binare, deoarece implică doar accesarea unui element indiciat. În cazul unui arbore binar de căutare aceste operaţii iau acelaşi timp ca o căutare obişnuită, adică `O(log n)` în cazul mediu şi `O(n)` în cazul cel mai nefavorabil.

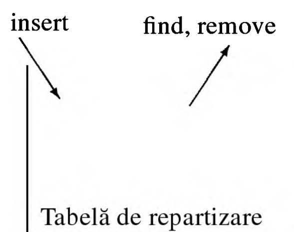
10.6 Tabele de repartizare

Există foarte multe aplicaţii care necesită o căutare dinamică bazată doar pe un nume. O aplicaţie clasică este *tabela de simboluri* a unui compilator. Pe măsură ce compilează programul, compilatorul trebuie să reţină numele (împreună cu tipul, durata de viaţă, locaţia de memorie) tuturor identificatorilor care au fost declaraţi. Atunci când vede un identificator în afara unei instrucţiuni de declarare, compilatorul verifică să vadă dacă acesta a fost declarat. Dacă a fost, compilatorul verifică informaţia adecvată din tabela de simboluri.

Având în vedere faptul că arborele binar de căutare permite acces logaritmic la obiecte cu denumiri oarecare, de ce am avea nevoie de o altă structură de date? Răspunsul este că arborele binar de căutare poate să dea un timp de execuţie liniar pentru accesul unui element, iar pentru a ne asigura de cost logaritmic este nevoie de algoritmi mult mai sofisticaţi.

Tabela de repartizare (engl. hashtable) este o structură de date care evită timpul de execuţie liniar, ba mai mult, suportă aceste operaţii în timp (mediu) constant. Astfel, timpul de acces la un element din tabelă nu depinde de numărul de elemente care sunt în tabelă. În acelaşi timp, tabela de repartizare nu foloseşte neapărat alocarea înlănţuită (ca arborele binar). Aceasta face ca tabela de repartizare să fie rapidă în practică. Un alt avantaj faţă de arborele binar de căutare este că elementele stocate în tabela de repartizare nu trebuie să implementeze interfaţa `Comparable`. Acum probabil că vă întrebaţi: bine, dar dacă tabela de repartizare este atât de eficientă, de ce se mai folosec arbori binari? Răspunsul este că arborii binari, deşi au operaţii de inserare şi acces care se execută în timp logaritmic dispun de operaţii pe care tabele de repartizare nu le suportă. Astfel, nu este posibil să parcurgem eficient elementele dintr-o tabelă de repartizare. Practic, tabelele de repartizare oferă un suport eficient pentru numai trei operaţii: adăgarea unui element, ştergerea unui element şi căutarea unui element. Parcurgerea elementelor, calculul minimului sau maximului nu sunt suportate.

Figura 10.15: Modelul pentru tabela de repartizare: orice element etichetat poate fi adăugat sau şters în timp practic constant.



Listing 10.23: Interfaţa pentru tabele de repartizare

```
package datastructures;

import exceptions.*;
/** Interfata expusa de o tabela de repartizare. Ofera
  * operatii de adaugare, stergere, cautare si golire. */
public interface HashTable
{
  void insert(Hashable x);
  void remove(Hashable x) throws ItemNotFoundException;
  Hashable find(Hashable x) throws ItemNotFoundException;
  void makeEmpty();
}
```

Operaţiile permise asupra unei tabele de repartizare sunt date în **Figura 10.15**, iar o interfaţă este prezentată în **Listing 10.23**. În acest caz, inserarea unui element care este duplicat nu generează o excepţie, deoarece elementul va fi înlocuit cu noua valoare. Aceasta este o alternativă la metoda pe care am aplicat-o în cazul arborilor binari de căutare. Tabela de repartizare funcţionează doar pentru elemente care implementează interfaţa `Hashable`[4]. Interfaţa `Hashable` (**Listing 10.24**) expune o *funcţie de repartizare*, care asociază obiectului de tip `Hashable` un număr întreg.

Fiecare dintre elementele conţinute de o tabelă de repartizare va fi asociat cu un *număr*, pentru a permite accesul în timp constant la elementul respectiv. Acest număr este calculat de metoda `hash()` expusă de interfaţa `Hashable`. Vom vedea mai târziu cum se face accesul la element pe baza acestui număr

[4]În limba engleză, tabelele de repartizare se numesc "Hashtable", de aceea un element care poate fi repartizat se numeşte "Hashable".

care este calculat într-un timp constant.

Elementele dintr-o tabelă de repartizare trebuie să redefinească şi metoda `equals()`. **Listing 10.25** arată cum clasa `MyString` implementează interfaţa `Hashable` prin implementarea metodelor `hash()` şi `equals()`.

Listing 10.24: Interfaţa `Hashable`

```
package datastructures;
/** Descrie un element care poate fi stocat intr-o
* tabela de repartizare */
public interface Hashable
{
  /** Asociaza un numar intreg instantei Hashable */
  int hash(int tableSize);
}
```

Listing 10.25: Clasa `MyString`

```
import datastructures.*;
import exceptions.*;

/** Exemplu de clasa care implementeaza interfata Hashable, deci
  * instantele ei pot fi stocate intr-o tabela de repartizare. */
public class MyString implements Hashable
{
  private String value;


  public MyString(String value)
  {
    this.value = value;
  }

  public String toString()
  {
    return value;
  }

  public boolean equals(Object c)
  {
    return value.equals( ((MyString) c).value);
  }

  public int hash(int tableSize)
  {
    return QuadraticProbingTable.hash(value, tableSize);
  }
}
```

Pentru a calcula numărul asociat unui obiect de tip `MyString` în cadrul

tabelei de repartizare, metoda `hash()` apelează la rândul ei o metodă statică a clasei `QuadraticProbingTable`, care va fi definită mai târziu (vezi **Listing 10.28**).

Listing 10.26: Un element al tabelei de repartizare

```
package datastructures;

/** Clasa care descrie un element dintr-o tabela de repartizare.
* Elementul este descris de o instanta Hashable si o valoare
* booleana care indica daca elementul este activ. */
class HashEntry
{
  Hashable element;
  boolean isActive;

  public HashEntry(Hashable e)
  {
    element = e;
    isActive = true;
  }

  public HashEntry(Hashable e, boolean i)
  {
    element = e;
    isActive = i;
  }
}
```

Interfaţa `HashTable` expune operaţiile pe care le suportă structura de date: inserare, ştergere, căutare şi eliminarea tuturor elementelor. Unele dintre aceste operaţii pot arunca excepţia `ItemNotFoundException`, care a fost prezentată în secţiunea 10.4.

Să vedem acum cum se implementează tabela de repartizare. Vom defini mai întâi un element al tabelei de repartizare, reprezentat de clasa `HashEntry` din **Listing 10.26**. `HashEntry` modelează un element din tabela de repartizare. Practic, tabela de repartizare va conţine elemente de tipul `HashEntry`. Pentru fiecare element din tabelă este necesar să specificăm valoarea lui (atributul `element`, de tip `Hashable`) şi dacă elementul este activ (atributul `isActive`). Trebuie menţionat că ştergerea unui element din tabela de repartizare nu implică eliminarea lui fizică din structură, ci doar "dezactivarea" lui, prin setarea atributului `isActive` la `false` (paragraful 10.6.1).

Implementarea propriu-zisă a tabelei de repartizare constă din următoarele două clase: `ProbingHashTable` şi `QuadraticProbingTable`.

Listing 10.27: Clasa `ProbingHashTable`

```
package datastructures;

import exceptions.*;
/** Implementeaza o tabela cu repartizare inchisa, in care
 * elementele tabelei de repartizare se retin intr-un tablou */
public abstract class ProbingHashTable
implements HashTable
{
  protected HashEntry[] elements;
  private int currentSize;
  private static final int DEFAULT_TABLE_SIZE = 11;

  /** Calculeaza pozitia unui element in cadrul tabelei*/
  protected abstract int findPos(Hashable x);

  /** Aloca memorie si intializeaza elementele tabloului */
  public ProbingHashTable()
  {
    elements = new HashEntry[DEFAULT_TABLE_SIZE];
    makeEmpty();
  }

  /** Goleste tabela de repartizare, si elibereaza toate
    * referintele din tabloul elements */
  public void makeEmpty()
  {
    currentSize = 0;

    for (int i = 0; i < elements.length; i++)
    {
      elements[i] = null;
    }
  }

  /** Intoarce elementul cu cheia x.hash()
    * @throws ItemNotFoundException daca elementul nu e gasit */
  public Hashable find(Hashable x)
  throws ItemNotFoundException
  {
    int currentPos = findPos(x);

    assertFound(currentPos, "Element negasit");

    return elements[currentPos].element;
  }

  /** Verifica daca elementul de pe pozitia currentPos
    * exista si este activ.
    * @throws ItemNotFoundException in caz contrar */
```

```
private void assertFound(int currentPos, String message)
throws ItemNotFoundException
{
  if (elements[currentPos] == null ||
    elements[currentPos].isActive == false)
  {
    throw new ItemNotFoundException(message);
  }
}

/** Sterge elementul x din tabela de repartizare.
  * @throws ItemNotFoundException daca x nu e gasit. */
public void remove(Hashable x)
throws ItemNotFoundException
{
  int currentPos = findPos(x);

  assertFound(currentPos, "Element negasit");

  elements[currentPos].isActive = false;
}

/** Adauga elementul x tabelei de repartizare. Daca
  * se atinge gradul de incarcare maxima adims se
  * dubleaza numarul de elemente din elements */
public void insert(Hashable x)
{
  int currentPos = findPos(x);

  elements[currentPos] = new HashEntry(x, true);

  if (++currentSize < elements.length / 2)
  {
    return;
  }

  //dimensiunea tabelei este prea mica
  //si pot aparea conflicte, deci vom dubla numarul de elemente
  HashEntry[] oldElements = elements;

  //creaza un tabel de dimensiune dubla
  elements = new HashEntry[oldElements.length] ;
  currentSize = 0;

  //copie vechiul tabel in cel nou
  for(int i = 0; i < oldElements.length; i++)
  {
    if (oldElements[i] != null && oldElements[i].isActive)
    {
      insert(oldElements[i].element);
```

```
      }
    }
  }

  /** Asociaza un numar unic unei chei de tip String
    * cu valoarea intre 0 si tableSize-1. */
  public static int hash(String key, int tableSize)
  {
    int hashVal = 0;

    for (int i = 0; i < key.length(); i++)
    {
      hashVal = 37 * hashVal + key.charAt(i);
    }

    hashVal %= tableSize;

    if (hashVal < 0)
    {
      hashVal += tableSize;
    }

    return hashVal;
  }
}
```

`ProbingHashTable` este clasa cea mai importantă în implementarea unei tabele de repartizare. Ea implementează interfaţa `HashTable` şi defineşte toate operaţiile expuse de aceasta. Elementele din tabela de repartizare sunt reţinute într-un şir de obiecte de tip `HashEntry` (atributul `elements`). Structura mai specifică de asemenea şi numărul de elemente care sunt stocate la momentul curent (atributul `currentSize`).

Inserarea unui element în tabela de repartizare presupune asocierea unui număr unic respectivului element, prin intermediul metodei `hash()`. Acest număr devine indicele elementului în şirul `elements`, motiv pentru care accesul în timp constant este asigurat. Practic, prin inserarea elementelor în tabela de repartizare, şirul `elements`, care reţine aceste elemente, va fi populat într-un mod discontinuu (elementele nu se vor afla pe poziţii succesive, pentru că poziţia lor depinde de valoarea lor). Când se va face căutarea elementului, se va realiza operaţia de `hash()` prin care se obţine numărul asociat, şi se va accesa direct elementul de pe poziţia calculată. Evident, accesul este în timp constant.

Metoda `hash()` calculează numărul asociat unui element pe baza unui algoritm simplu. Se realizează o sumă de produse pe baza codului *Unicode* al fiecărui caracter din cheia elementului care va fi inserat în tabelă, iar în final se consideră restul împărţirii acestei valori la dimensiunea tabelei, pentru că

Listing 10.28: Clasa `QuadraticProbingTable`

```
package datastructures;
/** Tabela cu repartizare patratica inchisa. Implementeaza metoda
* abstracta findPos() mostenita de la ProbingHashtable */
public class QuadraticProbingTable extends ProbingHashTable
{
  protected int findPos(Hashable x)
  {
    int collisionNum = 0;
    int currentPos = x.hash(array.length);

    while (array[currentPos] != null &&
      !array[currentPos].element.equals(x))
      {
        collisionNum++;
        currentPos += 2 * collisionNum - 1;

        if (currentPos >= array.length)
        {
          currentPos -= array.length;
        }
      }

      return currentPos;

  }
}
```

numărul asociat elementului trebuie să fie mai mic decât dimensiunea tabelei (numărul nu este altceva decât un indice al şirului `elements`).

10.6.1 Tratarea coliziunilor

Este important să observăm faptul că metoda `hash()` nu ne asigură de faptul că obiecte distince vor avea repartizări distincte. De fapt, există şanse considerabile ca pentru două elemente diferite, metoda `hash()` să returneze acelaşi număr. Aceasta este o situaţie inacceptabilă, pentru că nu putem avea două elemente care să fie pe aceeaşi poziţie în şirul `elements`. Din acest motiv, avem nevoie de o strategie care să elimine coliziunile care pot să apară. Strategia va determina dacă numărul asociat de metoda `hash()` este deja ocupat şi va oferi un alt număr care nu a fost alocat anterior. Această strategie va fi utilizată atât în operaţia de adăugare, cât şi în cea de căutare a elementului.

Există mai multe strategii disponibile, care se bazează pe căutarea unui spaţiu neocupat în şirul `elements`. De aici provine şi numele clasei `ProbingHashTable`. Cu alte cuvinte, în caz de coliziune, se *probează* elementele din şirul `elements`, până se descoperă unul neocupat.

Pentru a oferi suport pentru implementarea oricărei strategii, am decis ca `ProbingHashTable` să declare abstractă metoda care tratează situaţiile de coliziune (este vorba despre metoda `findPos()`), motiv pentru care şi clasa devine abstractă, urmând ca fiecare strategie să definească această metodă într-o clasă derivată separată.

Cea mai simplă strategie pentru evitarea coliziunii este *probarea* liniară, cu alte cuvinte căutarea secvenţială (element cu element) în şir, până se găseşte o celulă goală, care să fie alocată elementului în cauză. Performanţa acestei strategii nu este ridicată, mai ales în situaţia în care avem o tabelă de dimensiuni mari.

O strategie mai performantă este cea *pătratică*, în care căutarea nu se face liniar (`i, i + 1, i + 2, i + 3` etc.), ca în exemplul anterior, ci în "salturi" mai mari de tipul `i`, `i +` 1^2, `i +` 2^2, `i +` 3^2 etc. Acesta este motivul pentru care clasa care defineşte metoda `findPos()` poartă numele de `QuadraticProbingTable` (*quadratic* == pătratic). Deoarece în cazul unui conflict, elementul nu va fi pus pe poziţia dată de funcţia de repartizare ci pe locaţia liberă găsită prin probare, căutarea unui element va trebui făcută tot prin probare, deci tot utilizând metoda `findPos()`, apelată la linia 40 în cadrul metodei `find()` din clasa `ProbingHashTable`. Astfel, căutarea se va opri fie când elementul este găsit, fie când s-a ajuns la o locaţie goală în `elements`. O altă observaţie importantă este că ştergerea unui element nu va putea fi realizată prin setarea poziţiei care îi corespunde în `elements` la `null`, deoarece aceasta va împiedica găsirea elementelor cu aceeaşi valoare a funcţiei de repartizare care au fost aşezate prin probare după elementul şters. Acesta este motivul pentru care ştergerea unui element se face prin setarea atributului `isActive` la `false`.

Listing 10.29 prezintă un exemplu de utilizare al tabelei de repartizare.

Există foarte multe modalităţi de a implementa tabelele de repartizare. Noi am prezentat aici doar una singură, numită repartizare pătratică închisă. Pentru o detalii referitoare la acest subiect recomandăm excelenta lucrare *Data Structures* (vezi [Tomescu]).

Listing 10.29: Exemplu de program care foloseşte tabele de repartizare; programul va afişa: Gasit Marcel

```
import datastructures.*;
import exceptions.*;
```

```
/** Clasa simpla de test pentru tabelele de repartizare. */
public class TestHashTable
{
  public static void main(String[] args)
  {
    HashTable h = new QuadraticProbingTable();

    MyString result = null;

    h.insert(new MyString("Marcel"));

    try
    {
      result = (MyString) h.find(new MyString("Marcel"));
      System.out.println("Gasit " + result);
    }
    catch(ItemNotFoundException e)
    {
      System.out.println("Marcel nu a fost gasit");
    }
  }
}
```

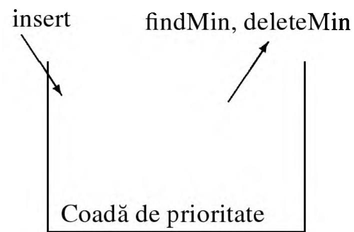
10.7 Cozi de prioritate

Să pornim de la o problemă concretă extrem de simplă: ordinea de tipărire a documentelor la o imprimantă partajată, în cadrul unei reţele, de un număr mare de utilizatori. În mod obişnuit documentele trimise unei imprimante sunt aşezate, într-o coadă simplă, dar aceasta nu este întotdeauna cea mai bună variantă. De exemplu, un document poate să fie deosebit de important, deci el ar trebui executat imediat ce imprimanta este disponibilă. De asemenea, dacă imprimanta a terminat de tipărit un document, iar în coadă se află câteva documente având 1-2 pagini şi un document având 100 de pagini, ar fi normal ca documentul lung să fie tipărit ultimul, chiar dacă nu este ultimul document trimis.

Analog, în cazul unui sistem multiutilizator, sistemul de operare trebuie să decidă la un moment dat care proces, dintre cele existente, trebuie să fie executat. În general, un proces poate să se execute doar o perioadă de timp fixată. Şi aici este normal ca procesele care au nevoie de un timp foarte mic să aibă prioritate.

Dacă vom atribui fiecărei sarcini câte un număr, atunci numărul mai mic (pagini tipărite, resurse folosite) va indica o prioritate mai mare. Astfel, vom dori să accesăm cel mai mic element dintr-o colecţie de elemente şi să îl ştergem

Figura 10.16: Modelul pentru coada de prioritate. Doar elementul minim este accesibil.



Listing 10.30: Interfata `PriorityQueue`

```
package datastructures;

import exceptions.*;
/**
 * Interfata care descrie operatiile expuse de o coada
 * de prioritati.
 */
public interface PriorityQueue
{
  /** Adauga elementul x in coada de prioritati*/
  void insert(Comparable x);
  /** Sterge si intoarce elementul minim. */
  Comparable deleteMin() throws UnderflowException ;
  /** Intoarce elementul minim din coada*/
  Comparable findMin() throws UnderflowException ;
  /** Goleste coada de prioritati*/
  void makeEmpty();
  /** Intoarce true daca coada este vida*/
  boolean isEmpty();

}
```

din cadrul colecţiei. Acestea sunt operaţiile `findMin` şi `deleteMin`. Structura de date care oferă o implementare foarte eficientă a acestor operaţii se numeşte *coadă de prioritate*. **Figura 10.16** ilustrează operaţiile fundamentale pentru coada de prioritate.

Interfaţa unei cozi de prioritate este descrisă în **Listing 10.30**.

Clasa de excepţii `UnderflowException` a fost prezentată în secţiunea 10.2 şi are rolul de a semnala situaţia în care s-a încercat accesarea unui element

Figura 10.17: Un arbore binar complet. Se observă că toate nodurile au exact doi fii, mai puţin nodurile de pe ultimul nivel (D, E, F şi G) pentru care frunzele (H, I şi J) sunt completate de la stânga la dreapta.

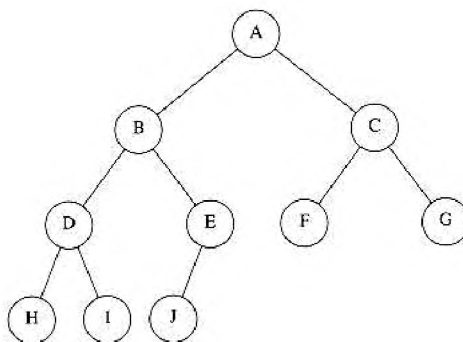


Figura 10.18: Reprezentarea arborelui din figura 10.17 utilizând un tablou. Nodurile arborelui sunt introduse în tablou în ordinea naturală a citirii lor de sus în jos şi de la stânga la dreapta. Prima locaţie rămâne necompletată.

	A	B	C	D	E	F	G	H	I	J			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

într-o coada de priorităţi goală.

10.7.1 Ansamble

Implementarea cozii de priorităţi o prezentăm sub forma unui ansamblu (engl. *heap*, tradus în unele lucrări prin termeni amuzanţi cum ar fi movilă sau grămadă). Ansamblele sunt arbori binari compleţi. Cu alte cuvinte, fiecare nod are câte doi fii, mai puţin pe ultimul nivel, unde se poate întâmpla ca un nod să nu aibă fii, dar în această situaţie, nici un nod aflat la dreapta sa nu va avea fii.

O observaţie importantă pe care o putem face este că dat fiind faptul că un arbore binar complet este atât de regulat, elementele sale pot fi cu uşurinţă reprezentate într-un şir, fără a fi necesar să utilizăm alocarea înlănţuită din paragraful 10.5.3! Tabloul din **Figura 10.17** corespunde arborelui din **Figura 10.18**.

Problema care se pune acum este: cum găsim în reprezentarea sub formă de şir a arborelui care sunt fii şi care este părintele unui anumit nod? Privind cu atenţie cele două figuri se observă cu uşurinţă că întotdeauna fii unui nod care este pe poziţia `i` în tablou se vor afla pe poziţiile `2*i` şi `2*i+1`. Pe poziţia `2*i` se află fiul stâng, iar pe poziţia `2*i+1` se află fiul drept. În consecinţă, părintele unui nod aflat de poziţia `i` se va afla pe poziţia `i/2` (împărţire întreagă). Astfel, nu numai că nu este nevoie de alocare înlănţuită pentru a reprezenta un ansamblu, ci, mai mult, operaţiile de traversare vor fi extrem de rapide (vezi capitolul 2 din primul volum, subcapitolul *Operatori la nivel de bit*)! Singura problemă cu această implementare este că avem nevoie de o aproximare a numărului maxim de elemente din ansamblu pentru a stabili dimensiunea şirului în care vor fi memorate elementele.

Pe lângă proprietatea structurală dată la începutul acestui paragraf, un ansamblu mai are şi o proprietate de ordonare care permite execuţia rapidă a operaţiilor caracteristice: inserare şi extragerea celui mai mare (sau a celui mai mic) element. Deoarece dorim să găsim rapid cel mai mare (mic) element, are sens să aşezăm cel mai mare (mic) element în rădăcină. Dacă considerăm că şi subarborii stâng şi drept trebuie să fie şi ei ansamble, atunci fiecare nod trebuie să fie mai mare (mic) decât descendenţii săi.

Aplicând acest demers logic, ajungem la proprietatea de ordonare a ansamblelor: valoarea oricărui nod este cel puţin la fel de mare ca valoarea fiilor săi. În **Figura 10.19**, arborele din stânga este un min-ansamblu, în timp ce arborele din dreapta nu este un ansamblu. Pentru ca arborele ansamblul să respecte această proprietate, indiferent de operaţiile care se realizează asupra lui, trebuie ca la operaţiile de inserare (metoda `insert()`) şi ştergere (metoda `deleteMin()`) să se realizeze o reordonare a nodurilor arborelui, astfel încât arborele modificat să respecte şi el proprietatea de ordine. Aceasta înseamnă propagarea elementelor înspre şi dinspre rădăcină, operaţie realizată de metoda `fixHeap()`, care filtrează toate elementele şirului (metoda `percolateDown()`).

Adăugarea unui element într-un ansamblu: Pentru a fixa ideile, vom conveni ca în continuare prin ansamblu vom înţelege un min-ansamblu. Inserarea unui element `x` în ansamblu va decurge astfel: vom crea un nod necompletat în următoarea locaţie disponibilă (căci altfel arborele nu ar fi complet). Dacă `x` poate fi plasat în noul nod fără a încălca proprietatea de ordonare, atunci suntem gata. Altfel, fie `y` părintele noului nod; deoarece `x` încalcă proprietatea de ordonare, avem evident x<y. Vom "împinge" pe `y` în jos, în locul noului nod inserat, urcând astfel nodul necompletat în sus în arbore. Continuăm acest proces de "împingere" până când noul element `x` poate fi plasat în nodul necompletat.

Figura 10.20 arată că pentru a insera valoarea 14 se creează un nod în urmă-

Figura 10.19: Doi arbori binari compleţi. Numai arborele din stânga este un ansamblu (un min-ansamblu). În arborele din dreapta nodul 40 are ca fiu nodul 34, ceea ce încalcă proprietatea de ordonare.

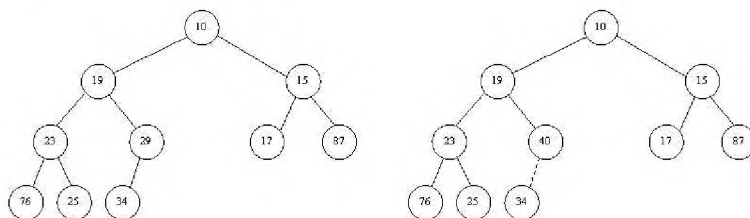
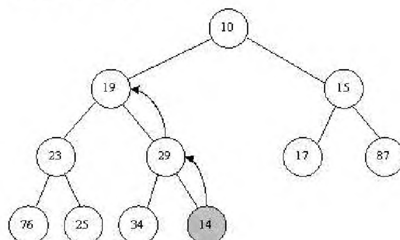


Figura 10.20: Inserarea nodului 14 în ansamblu: se creează o nouă locaţie, apoi această locaţie este "împinsă" în sus până ce nodul 14 poate fi aşezat în ea. În partea de jos este prezentată reprezentarea sub formă de şir utilizată de metoda `insert()`

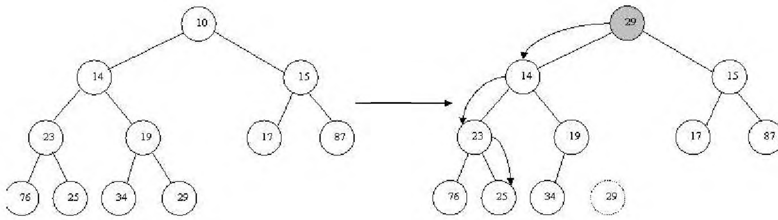


toarea locaţie disponibilă. Aşezarea lui 14 în noul nod ar încălca proprietatea de ansamblu, de aceea 29 "alunecă" în jos în locul noului nod. Paşii sunt continuaţi până când locul corespunzător pentru 14 este găsit. Denumirea în limba engleză pentru această modalitate de inserare este "percolate up" (filtrare în sus).

Metoda `insert()` din liniile 32-51 (**Listing** 10.31) realizează inserarea obiectului `x` de tip `Comparable` în ansamblu. În liniile 35-39 se verifică dacă tabloul `elements` chiar formează un ansamblu; dacă nu, pur şi simplu se adaugă `x` la finalul tabloului şi metoda se încheie. Dacă `elements` respectă proprietatea de ordine, atunci se verifică dacă mai este loc în tablou apelând `checkSize()` (linia 41) care dublează numărul de elemente dacă `elements` s-a umplut. Apoi se aplică filtrarea în sus (liniile 43-50), până când `x` ajunge la locul lui în ansamblu.

Timpul necesar pentru a insera un element în ansamblu poate ajunge până la $O(\log n)$ dacă elementul care trebuie inserat este chiar noul minim, deoarece

Figura 10.21: Crearea unei găuri în locul rădăcinii prin extragerea valorii minime. Din acest moment se caută locul potrivit pentru a insera ultimul nod, 29; se vor urca pe rând nodurile 14, 23, 25, după care se aşează 29 în locul lăsat liber de 25



va trebui filtrat în sus până ajunge la rădăcină (iar înălţimea ansamblului este $\log n$). În mod surprinzător, s-a arătat că în medie sunt necesare 2.6 comparaţii pentru a realiza o inserare, deci în medie un element urcă 1.6 niveluri. Aşadar, complexitatea medie a metodei `insert()` este $O(1)$.

Ştergerea unui element dintr-un ansamblu: Extragerea elementului minim dintr-un ansamblu este realizată foarte asemănător cu inserarea. Găsirea minimului este uşoară (acesta se află pe poziţia 1 în `elements`), partea mai dificilă este extragerea lui. Atunci când minimul este extras, se creează o "gaură" în locul rădăcinii. Deoarece ansamblul are acum cu un element mai puţin, rezultă că ultimul element, x, trebuie să urce undeva în sus în ansamblu. Dacă x poate fi plasat în locul "găurii" din rădăcină, atunci am terminat. Acest lucru este totuşi improbabil, de aceea "împingem" gaura în jos în locul fiului mai mic, care urcă în locul găurii. Repetăm acest pas până când x poate fi plasat în gaură.

Figura 10.21 arată un ansamblu înainte de ştergerea rădăcinii. După ce nodul 13 a fost extras, trebuie să plasăm nodul 31 undeva în ansamblu; 31 nu poate fi plasat în gaura creată, deoarece ar încălca proprietatea de ordonare. Astfel, vom plasa nodul 14 în locul rădăcinii, şi nodul gol alunecă un nivel în jos. Repetăm acest pas din nou, punând pe 19 în gaură şi avansând gaura şi mai adânc în arbore. Vom pune apoi pe 26 în gaură şi creăm o gaură în ultimul nivel. În sfârşit, vom putea pune nodul 31 în gaură. Această strategie este cunoscută sub numele de "filtrare în jos" (percolate down).

Ştergerea unui element din ansamblu este realizată de `deleteMin()` (liniile 145-154). La linia 147 se apelează metoda `findMin()` care întoarce elementul minim, sau aruncă o `UnderflowException` dacă ansamblul este vid (excepţia este aruncată mai departe de `deleteMin()`). În linia 149 este

aşezat în locul rădăcinii (elementul minim) elementul de pe ultima poziţie. Se apelează apoi metoda `percolateDown()` pentru filtra noua rădăcină şi a o aduce pe poziţia adecvată în ansamblu.

Filtrarea în jos (`percolateDown()`, liniile 113-140) a fost definită separat (spre deosebire de filtrarea în sus, care a fost inclusă în metoda `insert()`) deoarece ea va fi folosită în paragraful următor şi la refacerea ansamblului după adăugarea de elemente folosind `append()`. Metoda `percolateDown()` primeşte ca parametru poziţia pe care se află rădăcina (1 în cazul nostru). Ea presupune că atât subarborele stâng cât şi cel drept sunt ansamble şi doar rădăcina încalcă proprietatea de ordonare. La linia 116 se salvează rădăcina într-o variabilă temporară. Ciclul `while` de la liniile 118-137 realizează filtrarea propriu-zisă. În liniile 120-126 se calculează care este fiul mai mic al nodului `i` (care poate fi `2*i` sau `2*i+1`). Dacă rădăcina nu poate fi aşezată în locul fiului mai mic (linia 128) atunci fiul mai mic urcă în locul rădăcinii (linia 130), altfel ciclul `while` este întrerupt şi se plasează rădăcina pe poziţia `i` (linia 139). Ciclul `while` se execută până când fie ultimul element (salvat în `tmp`) poate fi aşezat în locul nodului curent, fie nodul curent nu mai are fii (şi deci putem adăuga rădăcina în locul lăsat liber de nodul curent).

Complexitatea în timp a algoritmului `deleteMin()` este proporţională cu adâncimea arborelui, adică $O(\log n)$. De obicei, gaura din rădăcină este coborâtă până în partea de jos a ansamblului, deoarece ultimul element este mare. Având în vedere faptul că adâncimea arborelui este $\log n$, rezultă că numărul mediu de execuţii ale corpului ciclului `while` este $O(\log n)$.

Construirea unui ansamblu în timp liniar: Adăugarea metodei `append()` la clasa `BinaryHeap` poate părea nejustificată. De ce să oferim o metodă care strică structura de ansamblu? Răspunsul este că extragerea elementului minim (care foloseşte de fapt proprietatea de ordonare) dintr-un ansamblu nu este realizată întotdeauna imediat ce am inserat un element. În multe situaţii practice, se inserează un număr nedeterminat de elemente şi abia după aceea se extrage elementul minim. Am văzut în paragrafele anterioare că ştergerea şi inserarea în ansamblu se fac în cazul cel mai nefavorabil în $O(\log n)$. Astfel, inserarea a n elemente într-un ansamblu iniţial gol s-ar face (în cazul cel mai nefavorabil) în $\log 1 + \log 2 + \ldots + \log n = \log n! \in O(n \log n)$. Desigur că ţinând cont de faptul că metoda `insert()` are totuşi o complexitate medie de $O(1)$, inserarea celor n elemente se face într-o complexitate medie de $O(n)$. Vom demonstra că folosind n apeluri ale metodei `append()` urmată de `fixHeap()` putem face acelaşi lucru în $O(n)$, chiar şi în cazul cel mai nefavorabil. Este uşor de văzut că metoda `append()` are complexitate $O(1)$, deci n apeluri ale ei vor fi realizate în $O(n)$. Vom arăta imediat că şi metoda `fixHeap()` are tot complexitate

$O(n)$, deci construirea ansamblului s-a realizat în $O(n)$.

Înainte de a analiza complexitatea în timp a metodei `fixHeap()`, să vedem cum funcţionează ea de fapt. Metoda `fixHeap()` va rearanja elementele tabloului `elements`, astfel încât ele să formeze un ansamblu. `fixHeap()` priveşte tabloul `elements` ca pe un arbore binar complet şi operează de la frunze către rădăcină. Să presupunem că tabloul `elements` conţine `n` obiecte. Subarborii de rădăcină `n/2+1`,`n/2+2`,...,`n` formează ansamble cu un singur nod, deci nu este necesar să fie modificaţi. Aşadar operarea începe descrescător, de la subarborele de rădăcină `n/2` până la cel de rădăcină 1, după cum se vede şi din ciclul `for` de la linia 103. Subarborele de rădăcină `n/2` poate fi transformat în ansamblu prin aplicarea algoritmului de filtrare în jos, deoarece atât subarborele stâng cât şi cel drept sunt deja ansamble. Analog se procedează pentru subarborele de rădăcină `n/2-1` până se ajunge la subarborele de rădăcină 1, care este chiar întreg ansamblul.

Să analizăm acum complexitatea în timp a metodei `fixHeap()`. În cazul cel mai nefavorabil, fiecare apel al metodei `percolateDown()` are complexitatea $h(i)$, unde $h(i)$ este înălţimea subarborelui de rădăcină i. Complexitatea metodei `fixHeap()` este aşadar proporţională cu suma înălţimilor nodurilor din ansamblu, care este $N - H - 1 \in O(n)$ în cazul unui ansamblu perfect de înălţime H (având $2^{H+1} - 1$ noduri).

Iată şi codul complet al clasei `BinaryHeap`:

Listing 10.31: Implementarea unei cozi de priorităţi cu ajutorul unui ansamblu

```
1 package datastructures;
2
3 import exceptions.*;
4
5 /** Implementarea unei cozi de prioritati folosind un
6 * ansamblu
7 */
8 public class BinaryHeap implements PriorityQueue
9 {
10   /** Numarul de elemente din ansamblu. */
11   private int currentSize;
12
13   /** true daca elements formeaza un ansamblu, false altfel */
14   private boolean orderOK;
15
16   /** Elementele din ansamblu. */
17   private Comparable[] elements;
18
```

```
/** Capacitatea implicita a ansamblului */
private static final int DEFAULT_CAPACITY = 11;

/** Contruieste un ansamblu, alocand elemente pentru elements
* si umpland pozitia 0 cu un element foarte mic */
public BinaryHeap(Comparable negInf)
{
  currentSize = 0;
  orderOK = true;
  elements = new Comparable[DEFAULT_CAPACITY];
  elements[0] = negInf;
}

/** Adauga elementul x in ansamblu */
public void insert(Comparable x)
{
  if (!orderOK)
  {
    append(x);
    return;
  }

  checkSize();

  int hole = ++currentSize;

  for ( ; x.compareTo(elements[hole / 2]) < 0; hole /= 2)
  {
    elements[hole] = elements[hole / 2];
  }

  elements[hole] = x;
}

/** Adauga un element ignorand proprietatea de ordonare */
public void append(Comparable x)
{
  checkSize();
  elements[++currentSize] = x;

  if (x.compareTo(elements[currentSize / 2]) < 0)
  {
    orderOK = false;
  }
}

/** Verifica daca sirul elements este plin, si in caz
* afirmativ ii dubleaza dimensiunea. */
private void checkSize()
{
```

```
    if (currentSize == elements.length - 1)
    {
      Comparable[] oldelements = elements;

      elements = new Comparable[currentSize * 2];

      for (int i = 0; i < oldelements.length; i++)
      {
        elements[i] = oldelements[i];
      }
    }
  }

  /** Intoarce elementul minim din ansamblu.
  * @throws UnserflowException Daca ansamblul este gol */
  public Comparable findMin() throws UnderflowException
  {
    if (isEmpty())
    {
      throw new UnderflowException("Heap vid");
    }

    if (!orderOK)
    {
      fixHeap();
    }

    return elements[1];
  }

  /** Transforma sirul elements intr-un ansamblu filtrand
  * toate nodurile care nu sunt frunze pana ajunge la radacina */
  private void fixHeap()
  {
    for (int i = currentSize / 2; i > 0; i--)
    {
      percolateDown(i);
    }

    orderOK = true;
  }

  /** Filtreaza elementul de pe pozitia hole asezandu-l in
  * subarborele cu radacina hole la locul cuvenit */
  private void percolateDown(int hole)
  {
    int child;
    Comparable tmp = elements[hole];

    while(hole * 2 <= currentSize)
```

```
    {
      child = hole * 2;

      if (child != currentSize &&
        elements[child + 1].compareTo(elements[child]) < 0)
      {
        child++;
      }

      if (elements[child].compareTo(tmp) < 0)
      {
        elements[hole] = elements[child];
      }
      else
      {
        break;
      }
      hole = child
    }

    elements[hole] = tmp;
  }

  /** Sterge elementul minim (aflat pe pozitia 1!) si
  * apeleaza algoritmul de filtrare pentru a reface ansamblul.
  * @throws UnderflowException daca ansamblul este vid. */
  public Comparable deleteMin() throws UnderflowException
  {
    Comparable minItem = findMin();

    elements[1] = elements[currentSize--];

    percolateDown(1);

    return minItem;
  }

  /** true daca ansamblul este vid */
  public boolean isEmpty()
  {
    return currentSize == 0;
  }

  /** Goleste ansamblul setand dimensiunea la 0*/
  public void makeEmpty()
  {
    currentSize = 0;
  }
}
```

Pentru a testa ansamblul, am creat următorul exemplu simplu:

Listing 10.32: Exemplu de utilizare a unei cozi de priorităţi. Programul va afişa: Continutul ansamblului: 0 1 2 3 4

```
import datastructures.*;
import exceptions.*;

/** Clasa de test simpla pentru cozi de prioritate. */
public class TestPriorityQueue
{
  public static void main(String[] args)
  {
    PriorityQueue pq = new BinaryHeap(
            new Integer(Integer.MIN_VALUE));

    pq.insert(new Integer(4));
    pq.insert(new Integer(2));
    pq.insert(new Integer(1));
    pq.insert(new Integer(3));
    pq.insert(new Integer(0));

    System.out.print("Continutul ansamblului: ");
    try
    {
      for ( ; ; )
      {
        System.out.print(pq.deleteMin() + " ");
      }
    }
    catch(UnderflowException e)
    {
    }
  }
}
```

Rezumat

Am prezentat pe scurt în cadrul acestui capitol cele mai importante structuri de date utilizate de programatori în practică. Fiecare structură de date oferă o interfaţă clară cu toate operaţiile pe care le permite, implementarea ei putând fi făcută în mai multe moduri. Stiva şi coada sunt structuri de date elementare, care oferă operaţii de adăugare respectiv ştergere la un singur capăt în timp constant. Listele înlănţuite permit operaţii de adăugare şi ştergere arbitrare şi au avantajul faţă de şiruri că inserarea unui element nu implică deplasarea elementelor aflate la dreapta. Arborii binari de căutare permit inserarea, cătuarea

şi ştergerea în timp logaritmic. Ei oferă de asemeni o modalitate de a parcurge în ordine elementele conţinute. Tabelele de repartizare permit o implementare extrem de eficientă a operaţiilor de inserare şi regăsire. În sfârşit, cozile de prioritate permit operaţii asupra elementului minim (sau maxim) din cadrul structurii în timp logaritmic. Ele au avantajul faţă de arborii binari (care permit şi ei acelaşi lucru) că sunt mai simplu de implementat şi nu implică reţinerea de legături căte fii din stânga şi dreapta. Structurile de date reprezinte o disciplină de sine stătătoare, iar noi nu am putut aici decât să oferim o introducere în acest domeniu. Cei care doresc să afle mai multe sunt invitaţi să parcurgă [Tomescu] şi [Cormen].

Începând cu următorul capitol, vor fi prezentate metodele fundamentale de elaborare a algoritmilor. Prima dintre ele este metoda căutării cu revenire (backtracking).

Noţiuni fundamentale

ansamblu binar: o variantă de arbore binar complet în care un nod este fie mai mic fie mai mare decât fii săi.

arbore: structură de date formată dintr-o mulţime de noduri şi muchii (folosite pentru a lega nodurile între ele) astfel încât nu există noduri care nu sunt legate direct sau indirect şi nu se pot realiza circuite.

arbore binar: arbore în care fiecare nod are cel mult doi fii.

arbore binar de căutare: variantă de arbore binar care suportă eficient operaţii de adăugare, ştergere şi căutare. Fiul din stânga este mai mic decât părintele, iar fiul din dreapta este mai mare.

clasă iterator: tip de clasă care permite manipularea unei liste.

coadă: structură de date care permite accesul doar la cel mai vechi element adăugat.

coadă de priorităţi: structură de date care permite accesul doar la elementul cel mai mic.

frunză: într-un arbore, reprezintă un nod fără fii.

funcţie hash: funcţie care calculează pe baza unui obiect primit, un întreg care va fi asociat cu obiectul respectiv. Doar obiectele care implementează interfaţa `Hashable` pot fi folosite de această funcţie.

stivă: structură de date care permite accesul doar la cel mai recent element inserat.

structură de date: o reprezentare a unor date şi a operaţiilor pe acele date. Prin intermediul structurilor de date se poate obţine reutilizarea componentelor.

tabelă de repartizare (hashtable): structură de date care permite adăugări, ştergeri şi căutări într-un timp mediu constant.

Erori frecvente

1. Se consideră eroare, încercarea de a accesa sau şterge elemente în cazul în care stiva, coada sau coada de priorităţi sunt vide.

2. O coadă de priorităţi nu are nici o legătură cu o coadă. Este doar o coincidenţă de nume.

3. Funcţia `hash()` pentru o tabelă de repartizare returnează o valoare de tip `int`. Este posibil ca în timpul calculelor intermediare să se producă depăşiri, aşadar este necesar să se verifice ca rezultatul operatorului modulo să fie pozitiv.

4. Performanţa unei tabele de repartizare se degradează semnificativ pe măsură ce factorul de încărcare se apropie de 1.0. Măriţi dimensiunea tabelei de îndată ce factorul de încărcare a atins valoarea 0.5.

Exerciţii

Pe scurt

1. Arătaţi care sunt rezultatele următoarei secvenţe de operaţii: `adaugă(6)`, `adaugă(12)`, `adaugă(2)`, `adaugă(7)`, `şterge()`, `şterge()` în cazul în care operaţiile `adaugă()` şi `şterge()` corespund operaţiilor de bază pentru:

   (a) stivă;

   (b) coadă;

   (c) arbore binar de căutare;

   (d) coadă de priorităţi.

2. Desenaţi arborii binari de căutare de înălţime 2,3,4,5 şi 6 pentru mulţimea de chei $\{1, 5, 8, 9, 12, 17, 29\}$!

3. Scrieţi ordinea nodurilor rezultată pentru parcurgerea în preordine, inordine şi postordine a arborilor de căutare de la exerciţiul precedent.

4. Să presupunem că avem numerele de la 1 la 1000 într-un arbore binar de căutare şi că dorim să căutăm numărul 363. Care dintre următoarele secvenţe nu poate constitui o secvenţă de noduri examinate?

   (a) 2, 252, 401, 398, 330, 344, 397, 363;

(b) 924, 220, 911, 244, 898, 258, 362, 363;

(c) 925, 202, 911, 240, 912, 245, 363;

(d) 2, 399, 387, 219, 266, 382, 381, 278, 363;

(e) 935, 278, 347, 621, 299, 392, 358, 363.

5. Desenaţi ansamblul care rezultă în urma inserării elementelor 47, 28, 19, 6, 15, 12, 3, 5, 9, 14 într-un ansamblu iniţial vid!

Teorie

1. Să presupunem că am implementa coada din paragraful 10.3 fără a aşeza elementele circular. Găsiţi o secvenţă de operaţii, care repetată la infinit conduce la mărirea nedefinită a tabloului `elements`, chiar dacă în coadă sunt doar câteva elemente.

2. Să presupunem că dorim o structură de date care să suporte *doar* următoarele operaţii: adăugare, găsirea maximului şi ştergerea maximului. Cât de eficient pot fi implementate aceste operaţii?
Indicaţie: Se utilizează un ansamblu în care rădăcina este mai mare decât fii.

3. Există o posibilitate de a implementa următoarele operaţii în timp logaritmic în cadrul aceleiaşi structuri de date: găsire minim şi maxim, ştergere minim şi maxim, adăugare?
Indicaţie: Se utilizează un arbore binar de căutare.

4. Există posibilitatea de a implementa următoarele operaţii în timp constant: `push`, `pop` şi găsirea minimului?
Indicaţie: Se observă că nu se cere şi ştergerea minimului. Putem astfel utiliza două stive, una care reţine elementele structurii de date, iar alta care reţine valorile minime.

5. De ce în cazul metodei `findMin()` din clasa `BinarySearchTree` nu a fost nevoie să salvăm valoarea parametrului `t`, care este modificat de către metodă?

6. Profesorul Ştietot are impresia că a descoperit o proprietate remarcabilă a arborilor de căutare. Să presupunem că o căutare pentru cheia `k` într-un arbore de căutarese termină într-o frunză. Considerăm următoarele mulţimi: A: nodurile din stânga drumului parcurs până la `k`, B: mulţimea nodurilor aflate chiar pe acest drum şi C, mulţimea cheilor din dreapta

drumului. Profesorul Ştietot susţine că toate cheile din A sunt mai mici decât cele din B, care sunt la rândul lor mai mici decât cele din C. Daţi cel mai mic contraexemplu care să spulbere afirmaţia lui Ştietot!

7. Care este numărul minim şi numărul maxim de elemente într-un ansamblu cu înălţimea h?

8. Arătaţi că un ansamblu cu n elemente are înălţimea $\lfloor \log_2 n \rfloor$!

9. Unde poate să se afle cel mai mare element într-un min-ansamblu?

10. Un tablou cu elementele ordonate crescător este un ansamblu?

11. Secvenţa 47, 28, 19, 6, 15, 12, 3, 5, 9, 14 este un ansamblu?

În practică

1. În paragraful 10.2 am văzut cum se poate implementa o stivă folosind un şir pentru a reţine elementele. Implementaţi acum stiva folosind clasa `LinkedList` din paragraful 10.4 şi un iterator special, care permite doar operaţiile specifice stivei.

2. În paragraful 10.3 am văzut cum se poate implementa o coadă folosind un şir pentru a reţine elementele. Implementaţi acum coada folosind clasa `LinkedList` din paragraful 10.4 şi un iterator special, care permite doar operaţiile specifice cozii.

3. Scrieţi o metodă care afişează elementele dintr-o listă în ordine inversă. Pentru aceasta definiţi un iterator care parcurge lista şi depune fiecare element într-o stivă. După ce lista a fost parcursă se vor scoate elementele din stivă până când aceasta se goleşte.

Proiecte de programare

1. În paragraful 10.6 am văzut cum se implementează o tabelă de repartizare utilizând un tablou pentru a reţine elementele. În cazul unui conflict (două elemente care sunt repartizate în acelaşi loc) utilizam o funcţie de repartizare pătratică pentru a găsi o poziţie liberă în care să inserăm noul element. Există şi o altă posibilitate de a gestiona conflictele: fiecare element al tabloului `elements` este de fapt o listă înlănţuită. Astfel, conflictele se vor rezolva aşezând elementele succesiv în cadrul listei înlănţuite. Scrieţi o clasă `ChainedHashtable` care implementează interfaţa `Hashtable` folosind metoda descrisă anterior.

11. Metoda Backtracking

Ab uno disce omnes (După unul îi poţi judeca pe toţi).

Vergiliu, Eneida

Prima metodă de elaborare a algoritmilor pe care o vom prezenta este *backtracking*. Această metodă este utilizabilă pentru un cadru larg de probleme, iar modul ei de aplicare este aproape algoritmic. Dintre toate metodele de elaborare prezentate, *backtracking* este considerată a fi cea mai elementară, deoarece ea se reduce la o parcurgere exhaustivă inteligentă a spaţiului de căutare şi necesită cele mai puţine adaptări ale formei generale pentru a fi aplicată în probleme concrete.

Discuţia asupra acestei metode începe cu o prezentare a cadrului în care ea poate fi utilizată şi a principiilor esenţiale care stau la baza ei. Vom da apoi schema generală de rezolvare a problemelor backtracking, urmată de câteva exemple clasice în care această metodă se aplică.

În cadrul acestui capitol vom prezenta:

- Care sunt problemele cărora li se poate aplica metoda *backtracking*;
- Ce sunt condiţiile interne şi condiţiile de continuare, şi cum influenţează acestea eficienţa unui algoritm *backtracking*;
- Care sunt cei patru paşi care se aplică stării iniţiale a problemei în vederea obţinerii soluţiilor;
- Care este schema generală de rezolvare a problemelor *backtracking* şi cum se particularizează ea pentru diverse probleme concrete;
- Exemple clasice de probleme care admit rezolvare prin această metodă.

11.1 Prezentare generală

În practică apar adeseori situaţii în care rezolvarea unei probleme se reduce în esenţă la determinarea unor vectori de forma:

$$x = (x_1, x_2, \ldots, x_n)$$

unde:

- fiecare componentă x_i aparţine unei *mulţimi finite* V_i;
- componentele vectorului x respectă anumite relaţii, numite *condiţii interne*, astfel încât x este o soluţie a problemei dacă şi numai dacă aceste condiţii sunt satisfăcute de componentele $x_1, x_2, \ldots, x_n$ ale vectorului.

Produsul cartezian $V_1 \times V_2 \times \ldots \times V_n$ se numeşte *spaţiul soluţiilor posibile*. Elementele acestui produs cartezian care respectă condiţiile interne se numesc *soluţii* ale problemei.

Exemplul 1: Fie două mulţimi de litere $V_1 = \{A, B, C\}$ şi $V_2 = \{M, N\}$. Se cere să se determine acele perechi (x_1, x_2) având proprietatea că dacă x_1 este A sau B, atunci x_2 nu poate fi N.

Rezolvarea problemei de mai sus conduce la perechile:

$$(A, M), (B, M), (C, M), (C, N)$$

deoarece din cele şase soluţii posibile, doar acestea îndeplinesc condiţiile puse în enunţul problemei.

Exemplul 2: Se dă mulţimea cu elementele $\{A, B, C, D\}$. Se cere să se genereze toate permutările elementelor acestei mulţimi. Se cer aşadar cvadrupletele de forma $x = (x_1, x_2, x_3, x_4)$ care respectă condiţiile

- $x_i \neq x_j$ pentru $i \neq j$;
- x_i aparţine mulţimii $V = V_i = \{A, B, C, D\}, \ i = 1, 2, 3, 4$.

Există mai mulţi vectori ce respectă aceste condiţii: $\{A, B, C, D\}$, $\{B, A, C, D\}$, $\{B, C, D, A\}$, $\{B, D, A, C\}$ etc. Mai exact, numărul de permutări ale elementelor unei mulţimi cu 4 elemente este 4! = 24.

O modalitate de rezolvare a problemei ar fi să se genereze toate cele 4^4 = 256 elemente ale produsului cartezian $V_1 \times V_2 \times V_3 \times V_4$ (reprezentând soluţiile

posibile) şi să se aleagă dintre ele cele 24 care respectă condiţiile interne. Să observăm însă că dacă în loc de 4 elemente mulţimea noastră ar avea 7 elemente, vor exista $7^7 = 823.543$ variante posibile, dintre care doar $7! = 5040$ (adică doar 0.6%) vor respecta condiţiile interne. Aceasta înseamnă că 99.4% dintre variante sunt generate inutil.

Din cele arătate mai sus reiese că are sens să ne punem problema de a găsi algoritmi mai eficienţi, care să evite generarea brutală a întregului spaţiu de soluţii pentru a selecta apoi soluţiile care respectă condiţiile interne. Metoda *backtracking* este o metodă foarte importantă de elaborare a algoritmilor pentru problemele de genul celor descrise mai sus. Deşi algoritmii de tip backtracking au şi ei, în general, complexitate exponenţială, sunt totuşi net superiori unui algoritm care generează toate soluţiile posibile.

11.2 Prezentarea metodei

Aşa cum am văzut în paragraful anterior, *metoda backtracking* urmăreşte să evite generarea tuturor soluţiilor posibile, reducând astfel drastic timpul de calcul.

Să vedem acum care este modalitatea prin care *backtracking* generează soluţiile, evitând parcurgerea exhaustivă a spaţiului de căutare. Componentele vectorului x primesc valori în ordinea crescătoare a indicilor (vom nota aceste valori cu $v_1, v_2, \ldots, v_n$ cu scopul de a face diferenţa între o componentă care nu are o valoare atribuită, x_k, şi o componentă care are atribuită o valoare, v_k). Aceasta înseamnă că lui x_k nu i se atribuie o valoare decât după ce $x_1, x_2, \ldots, x_{k-1}$ au primit valori. Mai mult decât atât, valoarea v_k atribuită lui x_k va trebui astfel aleasă încât $v_1, v_2, \ldots, v_k$ să respecte anumite condiţii, numite *condiţii de continuare*, care sunt deduse de către *programator* pe baza *condiţiilor interne*. Astfel, dacă în **Exemplul 2**, prima componentă, x_1, a primit valoarea $v_1 = A$, este clar că lui x_2 nu i se va mai putea atribui aceeaşi valoare (elementele unei permutări trebuie să fie diferite).

Neîndeplinirea condiţiilor de continuare exprimă faptul că oricum am alege valorile pentru componentele $x_{k+1}, \ldots, x_n$, nu vom obţine nici o soluţie (deci condiţiile de continuare sunt strict necesare pentru obţinerea unei soluţii). Prin urmare, se va trece la atribuirea unei valori lui x_k, doar dacă condiţiile de continuare pentru componentele $x_1, x_2, \ldots, x_k$ (având valorile $v_1, v_2, \ldots, v_k$) sunt îndeplinite. În cazul neîndeplinirii condiţiilor de continuare, se alege o nouă valoare pentru x_k sau, în cazul în care mulţimea valorilor posibile, V_k, a fost epuizată, se încearcă să se facă o nouă alegere pentru componenta *precedentă*, x_{k-1}, a vectorului, micşorând pe k cu o unitate. Această revenire la componenta

precedentă dă numele metodei, exprimând faptul că dacă nu putem avansa, urmărim (*engl.* track = "a urmări") înapoi (*engl.* back = "înapoi") secvenţa curentă din soluţie.

Trebuie observat faptul că respectarea condiţiilor de continuare de către valorile $v_1, v_2, \ldots, v_k$ nu reprezintă nicidecum o garanţie a faptului că vom obţine o soluţie continuând căutarea cu aceste valori. Deci condiţiile de continuare sunt *condiţii necesare* pentru ca $v_1, v_2, \ldots, v_k$ să conducă la o soluţie, dar nu sunt neapărat *condiţii suficiente*.

Alegerea condiţiilor de continuare este foarte importantă, o alegere bună ducând la o reducere substanţială a numărului de calcule. În cazul ideal, aceste condiţii ar trebui să fie nu numai *necesare*, ci chiar *suficiente* pentru obţinerea unei soluţii, ceea ce ar garanta că toate secvenţele $v_1, v_2, \ldots, v_k$ generate vor conduce în final la o soluţie. În practică acest lucru nu este adeseori posibil, de aceea se urmăreşte găsirea unor condiţii de continuare care să fie cât mai "dure", adică să elimine din start cât mai multe soluţii neviabile. De obicei, condiţiile de continuare reprezintă restricţia condiţiilor interne la primele k componente ale vectorului. Evident, condiţiile de continuare în cazul k=n sunt chiar condiţiile interne.

De exemplu, o condiţie de continuare în cazul problemei permutărilor din **Exemplul 2** ar fi:

$$v_k \neq v_i, \forall i = 1, k-1$$

Prin *metoda backtracking*, orice vector este construit *progresiv*, începând cu prima componentă şi mergând către ultima, cu eventuale *reveniri* asupra valorilor atribuite anterior. Reamintim că $x_1, x_2, \ldots, x_n$ primesc valori în mulţimile $V_1, V_2, \ldots, V_n$. Prin atribuiri sau încercări de atribuiri eşuate din cauza nerespectării condiţiilor de continuare, anumite valori sunt *consumate*. Pentru o componentă oarecare x_k vom nota prin C_k mulţimea valorilor consumate la momentul curent. Evident, $C_k \subset V_k$.

O descriere completă a stării în care se află un algoritm backtracking la un moment dat se poate face prin precizarea următoarelor elemente:

1. componenta din vector la care s-a ajuns în procesul de căutare, având valoarea k;

2. valorile $v_1, v_2, \ldots, v_{k-1}$ care au fost atribuite primelor $k-1$componente ale vectorului x;

3. mulţimile de valori $C_1, C_2, \ldots, C_k$ care au fost consumate pentru componentele $v_1, v_2, \ldots, v_{k-1}$ şi pentru componenta curentă, x_k.

Această descriere poate fi sintetizată într-un tabel numit *configuraţie*, având următoarea formă:

$$\left(\begin{array}{c|c} v_1, \ldots, v_{k-1} & x_k, x_{k+1}, \ldots, x_n \\ C_1, \ldots, C_{k-1} & C_k, \phi, \ldots, \phi \end{array} \right)$$

Semnificaţia unei astfel de configuraţii este următoarea:

1. în încercarea de a construi un vector soluţie a problemei, componentelor $x_1, x_2, \ldots, x_{k-1}$ li s-au atribuit valorile $v_1, v_2, \ldots, v_{k-1}$;
2. aceste valori satisfac condiţiile de continuare;
3. urmează să se atribuie o valoare componentei x_k; deoarece valorile consumate până în prezent sunt cele din mulţimea C_k, componenta x_k va putea primi o valoare v_k din $V_k - C_k$;
4. valorile consumate pentru componentele $x_1, x_2, \ldots, x_k$ sunt cele din mulţimile $C_1, C_2, \ldots, C_k$, cu precizarea că valorile curente $v_1, v_2, \ldots, v_{k-1}$ sunt consumate, deci apar în mulţimile $C_1, C_2, \ldots, C_{k-1}$;
5. pentru componentele $x_{k+1}, \ldots, x_n$ nu s-a încercat nici o atribuire, deci nu s-a consumat nici o valoare şi, prin urmare, $C_{k+1}, \ldots, C_n$ sunt vide;
6. până în acest moment au fost construite eventualele soluţii de forma:
   - $(c_1, \ldots)$ cu $c_1 \in C_1 - \{v_1\}$;
   - $(v_1, c_2, \ldots)$ cu $c_2 \in C_2 - \{v_2\}$;
   - .....
   - $(v_1, v_2, \ldots, v_{k-2}, c_{k-1}, \ldots)$ cu $c_{k-1} \in C_{k-1} - \{v_{k-1}\}$;
   - $(v_1, v_2, \ldots, v_{k-1}, c_k, \ldots)$ cu $c_k \in C_k$;

Această ultimă afirmaţie este mai dificil de înţeles şi recomandăm reluarea ei după lecturarea exemplului de mai jos.

În construirea permutărilor mulţimii cu elementele $\{A, B, C, D\}$ din **Exemplul 2**, pentru $k = 4$, configuraţia

$$\left(\begin{array}{ccc|c} C & A & B & x_4 \\ \{A, B, C\} & \{A\} & \{A, B\} & \{A, B\} \end{array} \right)$$

are, conform celor arătate mai înainte, următoarea semnificaţie:

1. componentele x_1, x_2, x_3 au primit respectiv valorile C, A, B;

2. tripletul C, A, B satisface condiţiile de continuare;

3. urmează să se atribuie o valoare componentei x_4. Componenta x_4 ia valori din mulţimea $V_4 - C_4$, adică una din valorile $\{C, D\}$;

4. $C_1 = \{A, B, C\}, C_2 = \{A\}, C_3 = \{A, B\}, C_4 = \{A, B\}$;

5. $k + 1 = 5 > n$, deci acest subpunct nu are obiect în această situaţie;

6. până în acest moment au fost deja construite soluţiile de forma (în ordinea în care au fost descrise la punctul 6 de mai sus):

   - $(A, \ldots)$ adică (A,B,C,D), (A,B,D,C), (A,C,B,D), (A,C,D,B), (A,D,B,C), (A,D,C,B) şi
     $(B, \ldots)$ adică (B,A,C,D), (B,A,D,C), (B,C,A,D), (B,C,D,A), (B,D,A,C), (B,D,C,A);
   - soluţii de această formă nu există, deoarece $C_2 - \{v_2\} = \phi$;
   - soluţii de forma $(C, A, A, \ldots)$: nu există soluţii de această formă;
   - soluţii de forma (C,A,B,A), (C,A,B,B): nu există soluţii de această formă.

Metoda backtracking începe a fi aplicată în situaţia în care nu s-a făcut nici o atribuire asupra componentelor lui x, deci nu s-a consumat nici o valoare, şi se încearcă atribuirea unei valori primei componente. Acest lucru este specificat prin *configuraţia iniţială*, a cărei formă este:

$$\left(\Big| \begin{array}{c} x_1, \ldots, x_n \\ \phi, \ldots, \phi \end{array} \right)$$

în care toate mulţimile C_k sunt vide.

Un alt caz special este cel al *configuraţiilor soluţie*, având forma:

$$\left(\begin{array}{c} v_1, \ldots, v_n \\ C_1, \ldots, C_n \end{array} \Big| \right)$$

cu semnificaţia că vectorul $(v_1, \ldots, v_n)$ este soluţie a problemei. Astfel, pentru **Exemplul 2**, configuraţia:

$$\left(\begin{array}{cccc} A & B & C & D \\ \{A\} & \{A, B\} & \{A, B, C\} & \{A, B, C, D\} \end{array} \Big| \right)$$

are semnificaţia că vectorul (A,B,C,D) constituie o soluţie a problemei.

Metoda backtracking constă în a porni de la configuraţia iniţială şi a-i aplica acesteia una dintre cele patru tipuri de transformări prezentate în continuare, până ce nu se mai poate aplica nici o transformare. Fiecare transformare se aplică în anumite condiţii bine precizate: la un moment dat doar o singură transformare poate fi aplicată. Presupunem că ne aflăm în configuraţia descrisă anterior, în care s-au atribuit valori primelor $k-1$ componente. Transformările care pot fi aplicate unei configuraţii sunt: atribuie şi avansează, încercare eşuată, revenire, revenire după construirea unei soluţii. Le vom prezenta pe fiecare pe rând.

11.2.1 Atribuie şi avansează

Acest tip de modificare are loc atunci când mai există valori neconsumate pentru componenta curentă, x_k, (deci $C_k \subset V_k$), iar valoarea aleasă, v_k, are proprietatea că $(v_1, \ldots, v_k)$ respectă condiţiile de continuare. În acest caz valoarea v_k se atribuie lui x_k şi se adaugă mulţimii C_k, după care se avansează la componenta următoare, x_{k+1}. Această modificare a configuraţiei poate fi reprezentată în felul următor:

$$\left(\begin{array}{c|c} \ldots, v_{k-1} & x_k, x_{k+1}, \ldots \\ \ldots, C_{k-1} & C_k, \quad \phi, \ldots \end{array}\right) \xrightarrow{v_k} \left(\begin{array}{c|c} \ldots, v_{k-1}, \qquad v_k & x_{k+1}, \ldots \\ \ldots, C_{k-1}, C_k \cup \{v_k\} & \phi, \ldots \end{array}\right)$$

De exemplu, la generarea permutărilor, avem următoarea schimbare de configuraţie pornind de la starea iniţială:

$$\left(\begin{array}{|cccc} x_1 & x_2 & x_3 & x_4 \\ \phi & \phi & \phi & \phi \end{array}\right) \xrightarrow{\text{A}} \left(\begin{array}{c|ccc} A & x_2 & x_3 & x_4 \\ \{\text{A}\} & \phi & \phi & \phi \end{array}\right)$$

11.2.2 Încercare eşuată

Acest tip de modificare are loc atunci când, ca şi în cazul anterior, mai există valori neconsumate pentru componenta curentă, x_k, dar valoarea v_k aleasă nu respectă condiţiile de continuare. În acest caz, v_k este adăugată mulţimii C_k (deci este consumată), dar *nu se avansează* la componenta următoare. Modificarea este notată prin:

$$\left(\begin{array}{c|c} \ldots, v_{k-1} & x_k, x_{k+1}, \ldots \\ \ldots, C_{k-1} & C_k, \quad \phi, \ldots \end{array}\right) \overset{v_k}{\widehat{===}} \left(\begin{array}{c|c} \ldots, v_{k-1} & x_k, \qquad x_{k+1} \ldots \\ \ldots, C_{k-1} & C_k \cup \{v_k\}, \phi, \ldots \end{array}\right)$$

În exemplul nostru cu generarea permutărilor, următoarea transformare este:

$$\left(\begin{array}{c|ccc} A & x_2 & x_3 & x_4 \\ \{A\} & \phi & \phi & \phi \end{array}\right) \overset{A}{\Longleftarrow} \left(\begin{array}{c|ccc} A & x_2 & x_3 & x_4 \\ \{A\} & \{A\}\phi & \phi & \end{array}\right)$$

11.2.3 Revenire

Acest tip de transformare apare atunci când toate valorile pentru componenta x_k au fost consumate ($C_k = V_k$), deci nu mai avem alte posibilităţi de a da valori acestei componente. În acest caz se revine la componenta precedentă, x_{k-1}, încercându-se atribuirea unei noi valori acestei componente. Este important de remarcat faptul că revenirea la x_{k-1} implică faptul că pentru x_k se vor încerca din nou toate variantele posibile, deci mulţimea C_k trebuie din nou să fie vidă. Transformarea este notată prin:

$$\left(\begin{array}{c|c} \ldots, v_{k-1} & x_k, x_{k+1}, \ldots \\ \ldots, C_{k-1} & C_k, \quad \phi, \ldots \end{array}\right) \longleftarrow \left(\begin{array}{c|c} \ldots, v_{k-2} & x_{k-1}, x_k, \ldots \\ \ldots, C_{k-2} & C_{k-1}, \quad \phi, \ldots \end{array}\right)$$

O situaţie de *revenire*, în exemplul cu generarea permutărilor este dată de configuraţia:

$$\left(\begin{array}{ccc|c} 3 & 1 & 2 & x_4 \\ \{1,2,3\} & \{1\} & \{1,2\} & \{1,2,3,4\} \end{array}\right) \longleftarrow \left(\begin{array}{cc|cc} 3 & 1 & x_3 & x_4 \\ \{1,2,3\} & \{1\} & \{1,2\} & \phi \end{array}\right)$$

11.2.4 Revenire după construirea unei soluţii

Acest tip de transformare se realizează atunci când toate componentele vectorului au primit valori care satisfac condiţiile interne, adică a fost găsită o soluţie. În această situaţie se revine din nou la cazul în care ultima componentă, x_n, urmează să primească o valoare.

Transformarea se notează astfel:

$$\left(\begin{array}{c|} \ldots, v_n \\ \ldots, C_n \end{array}\right) \overset{sol}{\longleftarrow} \left(\begin{array}{c|c} \ldots, v_{n-1} & x_n \\ \ldots, C_{n-1} & C_n \end{array}\right)$$

În exemplul nostru cu generarea permutărilor, revenirea după găsirea primei soluţii este dată de diagrama:

$$\left(\begin{array}{cccc|} 1 & 2 & 3 & 4 \\ \{1\} & \{1,2\} & \{1,2,3\} & \{1,2,3,4\} \end{array} \right) \xleftarrow{sol} \left(\begin{array}{ccc|c} 1 & 2 & 3 & x_4 \\ \{1\} & \{1,2\} & \{1,2,3\} & \{1,2,3,4\} \end{array} \right)$$

Revenirea după construirea unei soluţii poate fi considerată ca fiind un caz particular al *revenirii*, dacă adăugăm vectorului soluţie x o componentă suplimentară x_{n+1}, care nu poate lua nici o valoare ($V_{n+1} = \phi$).

O problemă importantă este cea a încheierii procesului de căutare a soluţiilor, sau, cu alte cuvinte, ne putem pune întrebarea: *transformările succesive aplicate configuraţiei iniţiale se încheie vreodată sau continuă la nesfârşit?* Evident că pentru ca *metoda backtracking* să constituie un algoritm trebuie să respecte proprietăţile unui algoritm, printre care se află şi proprietatea de *finitudine*. Demonstrarea finitudinii algoritmilor de tip backtracking se bazează pe următoarea observaţie simplă: prin transformările succesive de configuraţie nu este posibil ca o configuraţie să se repete, iar numărul de elemente al produsului cartezian $V_1 \times V_2 \times \ldots \times V_n$ este finit. Prin urmare, la un moment dat se va ajunge la configuraţia:

$$\left(\begin{array}{|cccc} x_1 & x_2 & \ldots & x_n \\ V_1 & \phi & \ldots & \phi \end{array} \right)$$

numită *configuraţie finală*. În configuraţia de mai sus ar trebui să aibă loc o revenire (deoarece toate valorile pentru prima componentă au fost consumate), adică o deplasare a barei verticale la stânga. Acest lucru este imposibil, şi algoritmul se încheie deoarece nici una din cele patru transformări nu poate fi aplicată. În practică, această încercare de a deplasa bara de pe prima poziţie ($k = 1$) pe o poziţie anterioară ($k = 0$) este utilizată pe post de condiţie de terminare a algoritmului.

Înainte de a trece la implementarea efectivă a *metodei backtracking* în pseudocod, să generăm diagramele de configuraţie pentru **Exemplul 1**:

$$\left(\Big|\begin{array}{cc} x_1 & x_2 \\ \phi & \phi \end{array}\right) \xrightarrow{\text{A}} \left(\begin{array}{c|c} A & x_2 \\ \{A\} & \phi \end{array}\right) \xrightarrow{\text{M}} \left(\begin{array}{cc} A & M \\ \{A\} & \{M\} \end{array}\Big|\right) \xleftarrow{sol} \left(\begin{array}{c|c} A & x_2 \\ \{A\} & \{M\} \end{array}\right) \overbrace{===}^{\text{N}}$$

$$\left(\begin{array}{c|c} A & x_2 \\ \{A\} & \{M,N\} \end{array}\right) \longleftarrow \left(\Big|\begin{array}{cc} x_1 & x_2 \\ \{A\} & \phi \end{array}\right) \xrightarrow{\text{B}} \left(\begin{array}{c|c} B & x_2 \\ \{A,B\} & \phi \end{array}\right) \xrightarrow{\text{M}} \left(\begin{array}{cc} B & M \\ \{A,B\} & \{M\} \end{array}\Big|\right)$$

$$\xleftarrow{sol} \left(\begin{array}{c|c} B & x_2 \\ \{A,B\} & \{M\} \end{array}\right) \overbrace{===}^{\text{N}} \left(\begin{array}{c|c} B & x_2 \\ \{A,B\} & \{M,N\} \end{array}\right) \longleftarrow \left(\Big|\begin{array}{cc} x_1 & x_2 \\ \{A,B\} & \phi \end{array}\right) \xrightarrow{\text{C}}$$

$$\left(\begin{array}{c|c} C & x_2 \\ \{A,B,C\} & \phi \end{array}\right) \xrightarrow{\text{M}} \left(\begin{array}{cc} C & M \\ \{A,B,C\} & \{M\} \end{array}\Big|\right) \xleftarrow{sol} \left(\begin{array}{c|c} C & x_2 \\ \{A,B,C\} & \{M\} \end{array}\right) \xrightarrow{\text{N}}$$

$$\left(\begin{array}{cc} C & N \\ \{A,B,C\} & \{M,N\} \end{array}\Big|\right) \xleftarrow{sol} \left(\begin{array}{c|c} C & x_2 \\ \{A,B,C\} & \{M,N\} \end{array}\right) \longleftarrow \left(\Big|\begin{array}{cc} x_1 & x_2 \\ \{A,B,C\} & \phi \end{array}\right)$$

11.3 Implementarea metodei backtracking

Procesul de obţinere a soluţiilor prin *metoda backtracking* este uşor de programat, deoarece la fiecare pas se modifică foarte puţine componente (indicele k, reprezentând poziţia barei, componenta x_k şi mulţimea C_k). Algoritmul (scris în pseudocod) de mai jos construieşte configuraţia iniţială, după care aplică una dintre cele patru transformări descrise în paragraful anterior, până când se ajunge la configuraţia finală (adică până când se încearcă o revenire de pe prima poziţie):

iniţializează (citeşte) mulţimile de valori, $V_1, \ldots, V_n$
$k \leftarrow 1$ *//se construieşte configuraţia iniţială*
pentru $i = 1, n$
    $C_i \leftarrow \phi$
//acum începe efectiv aplicarea celor 4 transformări, funcţie de caz
cât timp $k > 0$ *//*$k = 0$ *înseamnă terminarea căutării*
    dacă $k = n + 1$ *atunci* *//configuraţia este tip soluţie*
        reţine soluţia $v_1, \ldots, v_n$
        $k \leftarrow k - 1$ *//revenire după soluţie*
    altfel dacă $C_k \neq V_k$ *atunci* *//mai există valori neconsumate*

alege o valoare v_k *din* $C_k - V_k$
$C_k = C_k \cup v_k$ *//valoarea* v_k *este consumată*
dacă $v_1, \ldots, v_k$ *respectă condiţiile de continuare* *atunci*
$x_k \leftarrow v_k;$ *//atribuie şi*
$k \leftarrow k + 1;$ *//avansează*
altfel *//încercare eşuată, nu fac nimic*
altfel *//revenire*
$C_k \leftarrow \phi, k \leftarrow k - 1$
sfârşit cât timp

Algoritmul de mai sus funcţionează pentru cazul cel mai general, dar este destul de dificil de programat din cauza lucrului cu mulţimile C_k şi V_k. Din fericire, adeseori în practică mulţimile V_k au forma

$$V_k = \{1, 2, \ldots, s_k\}$$

deci fiecare mulţime V_k poate fi reprezentată foarte simplu prin numărul său de elemente, s_k. Pentru a simplifica şi mai mult lucrurile, în cadrul procesului de construire a unei soluţii vom alege valorile pentru fiecare componentă x_k în ordine crescătoare, pornind de la 1 şi până la s_k. În această situaţie, mulţimea de valori consumate C_k va fi de forma $\{1, 2, \ldots, v_k\}$ şi, drept consecinţă, va putea fi reprezentată doar prin valoarea v_k.

Consideraţiile de mai sus permit înlocuirea algoritmului anterior, bazat pe mulţimi, cu un algoritm simplificat care lucrează numai cu numere.

Dacă în **Exemplul 1** vom conveni să reprezentăm pe A, B, C prin valorile 1, 2, 3, iar pe M şi N prin 1 şi 2, configuraţiile care se succed în procesul de căutare pot fi reprezentate simplificat astfel:

$$(\ |\ x_1\, x_2\ ) \xrightarrow{1} (\ 1\ |\ x_2\ ) \xrightarrow[\text{etc}]{1} (\ 1\ \ 2\ |\ ) \xleftarrow{sol} (\ 1\ |\ x_2\ )$$

Particularizarea algoritmului pseudocod prezentat anterior se concretizează în următoarea metodă Java:

Listing 11.1: Metoda `backtracking`

```
public void backtracking()
{
    int k = 0;
    while (k >= 0)
    {
```

```
        if (k == n) //am gasit o solutie
        {
            retSol(); //afisam solutia
            k-- ; //revenire dupa gasirea unei solutii
        }
        else
        {
            if (x[k] < s[k]) //mai sunt valori neconsumate
            {
                x[k]++; //se ia urmatoarea valoare
                if (continuare(k)) //respecta cond. de cont?
                {
                    k++; //avanseaza
                }
            }
            else
            {
                x[k--] = 0; //revenire
            }
        }
    }
}
```

Metoda `backtracking()` foloseşte încă două metode:

- metoda `retSol()`, care, aşa cum sugerează şi numele ei, reţine soluţia, constând în valorile vectorului x. Cel mai adesea această metodă realizează o simplă afişare a soluţiei şi, eventual, o comparare cu soluţiile găsite anterior;
- metoda `continuare(k)` verifică dacă valorile primelor k componente ale vectorului x satisfac condiţiile de continuare; în cazul afirmativ este returnată valoarea `true`, iar în caz contrar este returnată valoarea `false`.

11.4 Probleme clasice rezolvabile prin backtracking

11.4.1 Problema generării permutărilor

Se dă mulţimea A cu elementele $\{a_1, a_2, \ldots, a_n\}$. Să se genereze toate permutările elementelor acestei mulţimi.

Se observă că această problemă este o simplă generalizare a **Exemplului 2** din acest capitol. Mai mult decât atât, problema poate fi redusă la a genera permutările mulţimii de indici $\{1, 2, \ldots, n\}$. În această situaţie vom avea

Listing 11.2: Funcţia de continuare pentru problema permutărilor

```
public boolean continuare(int k)
{
    for (int i = 0; i < k; ++i)
    {
        if (x[k] == x[i])
        {
            return false;
        }
    }
    return true;
}
```

$V_1 = V_2 = \ldots = V_n = \{1, 2, \ldots, n\}$, deci putem aplica varianta simplificată a *metodei backtracking*.

Condiţiile interne pe care trebuie să le respecte un vector soluţie sunt:

$$x_i \neq x_j \text{ pentru } \forall\, i, j = 1, n, \quad i \neq j.$$

Condiţiile de continuare pentru componenta numărul k sunt o simplă restricţie a condiţiilor interne:

$$x_i \neq x_k \text{ pentru } \forall\, i = 1, k-1.$$

Codul pentru funcţia de continuare este prin urmare foarte simplu, după cum reiese şi din **Listing 11.2**.

Metoda *backtracking* pentru generarea permutărilor se obţine din metoda *backtracking* pentru cazul general, înlocuind numărul de elemente al mulţimilor V_k(notat cu s_k) prin valoarea n. Soluţia completă a generării permutărilor este prezentată în **Listing 11.4**. Pentru a citi mulţimea de elemente care vor fi permutate am folosit clasa ajutătoare `Reader`, prezentată în volumul întâi, capitolul 4, paragraful *Variabila sistem CLASSPATH*. Reamintim încă o dată codul sursă al acestei clase.

Listing 11.3: Clasa `Reader`

```
package io;
//clasa va trebui salvata intr-un director cu numele "io"
//directorul in care se afla "io" va trebui adaugat
//in CLASSPATH
import java.io.*;
import java.util.StringTokenizer;
public class Reader
{
```

```
    public static String readString()
    {
      BufferedReader in = new BufferedReader(
      new InputStreamReader(System.in));
      try
      {
        return in.readLine();
      }
      catch(IOException e)
      {
        //ignore
      }
      return null;
    }

    public static int readInt()
    {
      return Integer.parseInt(readString());
    }

    public static double readDouble()
    {
      return Double.parseDouble(readString());
    }

    public static char readChar()
    {
      BufferedReader in = new BufferedReader(
      new InputStreamReader(System.in));
      try
      {
        return (char)in.read();
      }
      catch(IOException e)
      {
        //ignore
      }
      return '\0';
    }

    public static int[] readIntArray()
    {
      String s = readString();
      StringTokenizer st = new StringTokenizer(s);
      //aloca memorie pentru sir
      int[] a = new int[st.countTokens()];

      for (int i = 0; i < a.length; ++i)
      {
        a[i] = Integer.parseInt(st.nextToken());
```

```
      }

      return a;
    }

  }
```

Prin rularea programului următor se vor genera permutările mulţimii introduse de la tastatură.

Listing 11.4: Rezolvarea problemei permutărilor

```
import io.Reader;

/**
* Program care genereaza permutarile unei multimi
* introduse de la tastatura.
*/
public class Permutari
{
  /** Testeaza daca elementul adaugat exista deja.*/
  public static boolean continuare(int[] x, int k)
  {
    for (int i = 0; i < k; ++i)
    {
      if (x[k] == x[i])
      {
        return false;
      }
    }

    return true;
  }

  /** Construieste un string care contine solutia curenta.*/
  public static void retSol(int[] s, int[] x, int nrSol)
  {
    System.out.print("Permutarea " + nrSol + ": ");
    for (int i = 0; i < s.length; i++)
    {
      System.out.print(s[x[i] - 1] + " ");
    }
    System.out.println();
  }

  /**
  * Backtracking standard pentru determinarea
  * permutarilor multimii.
  */
  public static void backtracking(int[] s)
  {
```

```java
    int k = 0;
    //aloca memorie pentru sirul de indici
    int[] x = new int[s.length];
    int nrSol = 0;

    //initializeaza x
    for (int i = 0; i < x.length; i++)
    {
      x[i] = 0;
    }

    //procesul de backtracking
    while (k >= 0)
    {
      if (k == x.length) //am gasit o solutie
      {
        retSol(s,x,++nrSol); //afiseaza solutia
        k--; //revenire dupa ce o solutie a fost gasita
      }
      else
      {
        if (x[k] < x.length) //valori neconsumate?
        {
          //se ia urmatoarea valoare neconsumata
          x[k]++;

          //respecta valoarea aleasa
          //conditia de continuare?
          if (continuare(x, k))
          {
            k++; //avanseaza
          }
        }
        else
        {
          x[k--] = 0; //revenire
        }
      }
    }
  }

  /** Programul principal.*/
  public static void main(String[] args)
  {
    //citirea elementelor multimii
    System.out.println("Introduceti elementele multimii " +
    "(pe o linie, separate prin spatiu):");
    int[] s = Reader.readIntArray();

    //generarea permutarilor multimii
```

```
      backtracking(s);
    }
}
```

11.4.2 Generarea aranjamentelor şi a combinărilor

Vom vedea acum cât de simplu se poate adapta algoritmul de generare a permutărilor unei mulţimi pentru a genera aranjamentele şi combinările acelei mulţimi. Pentru a simplifica lucrurile, vom presupune că mulţimea A este formată din primele n numere naturale, adică $A = \{1, 2, \ldots, n\}$.

Reamintim faptul că prin aranjamente de n luate câte m ($n \geq m$), notate A_n^m se înţeleg toate mulţimile ordonate cu m elemente formate din elemente ale mulţimii A, cu alte cuvinte toţi vectorii de forma:

$$x = (x_1, \ldots, x_m), \text{ unde } x_i \in \{1, 2, \ldots, n\}, \quad x_i \neq x_j, \quad \forall i, j = 1, m$$

Se observă că, din punct de vedere al reprezentării formale, singura diferenţă dintre aranjamente şi permutări este că aranjamentele au lungime m în loc de n. De altfel, pentru m=n aranjamentele şi permutările coincid.

Exemplu: Aranjamentele de 3 luate câte 2 (A_3^2) sunt:

(1,2), (1,3), (2,1), (2,3), (3,1), (3,2).

Condiţiile interne şi, în consecinţă, condiţiile de continuare sunt identice cu cele de la generarea permutărilor. Prin urmare şi funcţia de continuare este identică cu cea de la permutări. Unde este totuşi diferenţa? Având în vedere că lungimea vectorului este m şi nu n, condiţia de găsire a unei soluţii trebuie adaptată. Prin urmare, în metoda `backtracking()` linia:

```
if (k == n)
```

va fi înlocuită cu:

```
if (k == m)
```

Desigur, aceeaşi modificare este necesară şi în metoda `retSol()`, în care secvenţa

```
for (int i = 0; i < n; ++i)
```

se va înlocui cu

```
for (int i = 0; i < m; ++i)
```

Listing 11.5: Funcţia de continuare pentru combinări

```
public boolean continuare(int k)
{
    if (k > 0 && x[k] <= x[k - 1])
    {
        return false;
    }
    else
    {
        return true;
    }
}
```

Să vedem acum modalitatea de generare a combinărilor. Reamintim că prin combinări de n luate câte m (notat C_n^m) se înţeleg toate submulţimile cu m elemente ale mulţimii $A = \{1, 2, \ldots, n\}$.

Exemplu: Combinările de 3 luate câte 2 (C_3^2) sunt:

(1,2), (1,3), (2,3).

Diferenţa între combinări şi aranjamente este dată de faptul că, în cazul combinărilor, ordinea în care apar componentele nu contează (combinarea (1,2) este aceeaşi cu combinarea (2,1) etc). Din acest motiv am optat în exemplul de mai sus să aranjăm componentele unei combinări în ordine crescătoare). Prin urmare, combinările unei mulţimi cu n elemente luate câte m sunt definite de vectorii:

$$x = (x_1, \ldots, x_m), \text{ unde } x_1 < x_2 < \ldots < x_m.$$

Condiţia de continuare în cazul combinărilor va fi pur şi simplu:

$$x_k > x_{k-1} \text{ pentru } k > 1.$$

Metodele `backtracking()` şi `retSol()` sunt, în cazul combinărilor, identice cu cele de la aranjamente. Diferenţa apare la funcţia de continuare, descrisă în **Listing 11.5**. **Listing 11.6** prezintă varianta mai elegantă a aceleiaşi funcţii.

Problema 3 de la finalul capitolului propune o variantă mai eficientă de generare a combinărilor în care funcţia de continuare este complet eliminată.

Listing 11.6: Funcţia de continuare pentru combinări (varianta elegantă)

```
public boolean continuare(int k)
{
    return k == 0 || x[k] > x[k - 1];
}
```

Figura 11.1: Soluţie de aşezare a damelor pe tabla de şah

	X		
			X
X			
		X	

11.4.3 Problema damelor

Să se aşeze n dame pe o tablă de şah de dimensiune n×n astfel încât damele să nu fie pe aceeaşi linie, aceeaşi coloană sau aceeaşi diagonală (damele să nu se atace între ele).

Reamintim că în jocul de şah, o damă "atacă" poziţiile aflate pe aceeaşi linie sau coloană şi pe diagonală. O posibilă aşezare a damelor pe o tablă de şah de dimensiuni 4×4 este dată în **Figura 11.1**.

Să vedem cum putem reformula problema damelor pentru a o aduce la o problemă de tip *backtracking*. Se observă cu uşurinţă că pe o linie a tablei de şah se poate afla o singură damă, prin urmare putem conveni că prima damă se va aşeza pe prima linie, a doua damă pe a doua linie etc. Rezultă că pentru a cunoaşte poziţia damei numărul *k* este suficient să ştim coloana pe care aceasta se găseşte. O soluţie a problemei se poate astfel reprezenta printr-un vector

$$x = (x_1, x_2, \ldots, x_n), \quad x_k \in \{1, 2, \ldots, n\},$$

unde x_k reprezintă coloana pe care se găseşte dama numărul *k*.

Cu această notaţie, vectorul soluţie corespunzător exemplului din **Figura 11.1** este: $(2, 4, 1, 3)$.

Să vedem acum care este condiţia ca două dame distincte, *k* şi *i*, să se atace:

- în mod cert damele nu pot fi pe aceeaşi linie;
- damele sunt pe aceeaşi coloană dacă $x_k = x_i$;

Listing 11.7: Funcţia de continuare pentru problema damelor

```
public boolean continuare(int k)
{
    for (int i = 0; i < k; ++i)
    {
        if (x[i] == x[k] || k-i == Math.abs(x[k]-x[i]))
        {
            return false;
        }
    }
    return true;
}
```

- damele sunt pe aceeaşi diagonală dacă se află pe colţurile unui pătrat, adică lungimea ($|x_k - x_i|$) este egală cu lăţimea ($k - i$):

$$|x_k - x_i| = |k - i|$$

Condiţia de continuare este ca dama curentă, k, să nu atace nici una dintre damele care deja sunt aşezate pe tablă, adică:

$$x_k \neq x_i \text{ şi } |x_k - x_i| \neq |k - i| \text{ pentru } \forall\, i = 1, k-1.$$

Transpusă în Java, funcţia de continuare are forma din **Listing 11.7.**

Modificările care trebuie aduse metodelor `retSol` şi `backtracking` sunt minime şi le lăsăm ca exerciţiu.

Observaţie: Problema damelor este primul exemplu de problemă în care condiţiile de continuare sunt necesare, dar nu sunt suficiente. De exemplu (pentru n=4), la început, algoritmul va aşeza prima damă pe prima coloană, a doua damă pe a treia coloană, iar cea de-a treia damă nu va putea fi aşezată pe nici o poziţie, fiind necesară o revenire.

11.4.4 Problema colorării hărţilor

Se dă o hartă ca cea din figura de mai jos, în care sunt reprezentate schematic 6 ţări, dintre care unele au graniţe comune. Presupunând că dispunem doar de trei culori (roşu, galben, verde), se cere să se determine toate variantele de colorare a hărţii astfel încât oricare două ţări vecine (care au frontieră comună) să fie colorate diferit.

Figura 11.2: Matricea de vecinătăţi corespunzătoare hărţii din partea stângă

T_2
T_1 T_3 T_4 T_5
T_6

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 0 \end{pmatrix}$$

Pentru a memora relaţia de vecinătate între două ţări vom utiliza o matrice de dimensiuni 6×6, numită *vecin*, definită prin:

$$vecin[i,j] = \begin{cases} true & \text{dacă ţările } T_i \text{ şi } T_j \text{ sunt vecine} \\ false & \text{altfel} \end{cases}$$

Figura 11.2 reprezintă matricea de vecinătăţi pentru harta cu 6 ţări în care s-a făcut convenţia că 1 reprezintă *true* şi 0 reprezintă *false*.

Problema se poate generaliza uşor şi la o hartă cu *n* ţări care trebuie colorată cu *m* culori. Vom utiliza pentru uşurarea expunerii harta cu 6 ţări de mai sus.

În această problemă, un vector soluţie $x = (x_1, x_2, \ldots, x_n)$ reprezintă o variantă de colorare a hărţii, având semnificaţia că ţara numărul *i* va fi colorată cu culoarea x_i. În exemplul nostru, x_i poate fi 1,2 sau 3, corespunzând respectiv culorilor roşu, galben, verde.

Condiţia de continuare este ca ţara căreia urmărim să îi atribuim o culoare, să aibă o culoare distinctă de ţările cu care are graniţă. Cu alte cuvinte, trebuie să avem $x_i \neq x_k$ dacă $A[i,k] = 1, \quad \forall i = 1, k-1$.

Funcţia de continuare care descrie condiţia de mai sus este dată în **Listing 11.8**.

Pentru o mai bună înţelegere a mecanismului *metodei backtracking* aplicată la problema colorării hărţilor, putem să ne imaginăm că dispunem de 6 cutii identice $V_1, V_2, \ldots, V_6$, fiecare dintre cutii conţinând trei creioane colorate notate cu r - roşu, g - galben, v - verde. Fiecare cutie V_k conţine creioanele care pot fi utilizate pentru colorarea ţării T_k.

O vizualizare a procesului de căutare a soluţiilor poate fi obţinută dacă aranjăm cele 6 cutii în ordine (fiecărei ţări îi asociem o cutie) şi punem un semn înaintea cutiei din care urmează să se aleagă un creion (marcajul corespunde barei verticale de la configuraţii); iniţial acest semn este în stânga primei cutii. Atunci când se alege un creion dintr-o cutie corespunzătoare unei ţări el va fi aşezat fie pe ţara respectivă, dacă nu există o ţară vecină cu aceeaşi culoare,

Listing 11.8: Funcţia de continuare pentru problema colorării hărţilor

```
public boolean continuare(int k)
{
    for (int i = 0; i < k; ++i)
    {
        if (x[i] == x[k] && vecin[k][i] == 1)
        {
            return false;
        }
    }
    return true;
}
```

fie lângă cutie în caz contrar. Astfel, mulţimile C_i de valori consumate la un moment dat sunt alcătuite din creioanele de lângă cutia V_i şi de pe ţara T_i. Cu aceste precizări, cele 4 modificări de configuraţie au următoarele semnificaţii concrete:

- *atribuie şi avansează*: se aşează creionul ales pe ţara corespunzătoare şi se trece la cutia următoare;
- *încercare eşuată*: creionul ales este aşezat lângă cutia din care a fost scos;
- *revenire*: creioanele corespunzătoare ţării curente sunt repuse în totalitate la loc în cutie şi se trece la cutia precedentă;
- *revenire după găsirea unei soluţii*: semnul este adus la stânga ultimei cutii.

Procesul se încheie în momentul în care toate creioanele ajung din nou în cutiile în care se aflau iniţial.

Rezumat

În acest capitol am prezentat *metoda bactracking*, care se reduce în esenţă la parcurgerea exhaustivă a spaţiului de căutare, în care se elimină cu grijă configuraţiile care nu pot conduce la o soluţie. Metoda *backtracking* se aplică oricărei probleme a cărei soluţie se poate scrie sub formă de şir, cu fiecare e-lement al şirului luând valori în cadrul unei mulţimi finite. Elementul estenţial care determină eficienţa căutării este reprezentat de *condiţiile de continuare* care

sunt utilizate pentru a reduce dimensiunile spaţiului de căutare. În modelul teoretic al acestei metode, se porneşte de la o configuraţie iniţială, căreia i se aplică succesiv anumite transformări până în momentul în care se ajunge la configuraţia finală. În practică, rezolvarea unei probleme prin această metodă se reduce cel mai adesea la scrierea funcţiei de continuare (care este o implementare a *condiţiilor de continuare*) pentru problema concretă căreia i se aplică.

Noţiuni fundamentale

backtracking: metodă de elaborare a algoritmilor care constă în construirea soluţiei componentă cu componentă, cu eventuale reveniri asupra componentelor anterioare.

condiţie internă: condiţia pentru ca un şir să reprezinte o soluţie a problemei.

condiţie de continuare: condiţie necesară pentru ca un şir parţial să poată conduce la o soluţie.

configuraţie finală: configuraţie căreia nu i se mai poate aplica nici una dintre cele patru transformări şi care indică încheierea procesului de căutare.

configuraţie iniţială: configuraţia de la care se porneşte în procesul de căutare al soluţiilor.

diagramă de stare: diagramă care sintetizează starea în care se află procesul de căutare la un anumit moment.

soluţie: un şir care respectă condiţiile interne.

Erori frecvente

1. Există adeseori situaţii în care soluţia unei probleme nu se poate scrie direct sub formă de vector, ci trebuie făcute anumite convenţii pentru a o reduce la un vector. Aşadar, *backtracking* se poate aplica oricărei probleme a cărei soluţie se poate *reduce* sau transforma într-un vector. Mulţi programatori începători renunţă la a încerca să aplice această metodă dacă soluţia ei nu este în mod evident structurată sub formă de vector, fără a-şi pune problema de a transforma soluţia într-o astfel de formă (vezi problema damelor din paragraful 11.4.3, în care soluţia se reduce de la o matrice (tabla de şah) la un şir de numere).

2. Deşi metoda *backtracking* se *poate* aplica oricărei probleme a cărei soluţie se poate scrie sub formă de şir, aceasta nu înseamnă că *backtracking* este şi metoda cea mai eficientă de a o rezolva. De exemplu, şi în cazul problemei sortării soluţia se scrie sub formă de şir, deci se poate aplica *backtracking*. Totuşi aplicarea lui *backtracking* în această situaţie ar însemna

o generare optimizată a permutărilor elementelor acelui şir, ceea ce ar conduce la o soluţie exponenţială, deci inutilizabilă. O să prezentăm în capitolul următor metode eficiente de ordonare a unui şir.

3. Adeseori funcţia de continuare nu este optim aleasă, în aşa fel încât să elimine cât mai multe configuraţii nefezabile.

4. Se confundă adeseori dimensiunea spaţiului de căutare (notată de noi cu n) cu dimensiunea individuală a fiecărei mulţimi V_k (notată de noi su $s[k]$). Este adevărat ca la multe probleme (de exemplu, permutări) acestea sunt egale, dar la altele (cum ar fi colorarea hărţilor, combinări, aranjamente), ele diferă.

Exerciţii

Teorie

1. Enumeraţi problemele prezentate în cadrul acestui capitol, pentru care condiţiile de continuare sunt necesare, dar nu sunt suficiente.

2. Luaţi o tablă de şah obişnuită şi încercaţi să simulaţi modul în care se generează soluţia problemei damelor pentru $n = 8$.

3. Găsiţi toate soluţiile de colorare cu trei culori a hărţii din **Figura 11.2**.

În practică

1. Să se afişeze toate modurile în care *n* persoane pot fi aşezate la o masă rotundă precum şi numărul acestora.

   Indicaţie: Există două posibilităţi de rezolvare:

   (a) *Se vor genera toate permutările posibile, prin metoda backtracking, şi se vor contoriza. Se va afişa apoi numărul lor. Va trebui însă să ţineţi seama de faptul că unele dispuneri sunt identice din cauza mesei circulare*;

   (b) *Mult mai elegant, folosind o observaţie simplă. Astfel, cu n obiecte se pot forma* $n!$ *permutări. Cum, în cazul dispunerii lor circulare* $1, 2, \dots, n$, *respectiv* $2, 3, \dots, n, 1$; ...; $n, 1, 2, \dots, n-1$ *sunt identice, rezultă că din* n *astfel de permutări trebuie considerată doar cea care începe cu 1. Numărul de permutări va fi aşadar* $\frac{n!}{n} = (n-1)!$.

2. Idem problema 1, cu precizarea că anumite persoane nu se agreează, deci nu pot fi aşezate una lângă cealaltă. La intrare se mai furnizează o matrice simetrică A, cu următoarea semnificaţie:

$$A(i,j) = \begin{cases} 1 & \text{dacă nu se agrează} \\ 0 & \text{altfel} \end{cases}$$

3. Să se modifice algoritmul de generare a combinărilor prezentat în paragraful 11.4.2 astfel încât funcţia de continuare să nu mai fie necesară.

   Indicaţie: Pentru fiecare componentă $x[k]$ se porneşte cu valoarea $x[k-1]+1$.

4. Se dau *n* mulţimi $A_1, A_2, \ldots, A_n$. Să se afişeze produsul lor cartezian.

   Indicaţie: Generarea produsului cartezian înseamnă de fapt generarea întregului spaţiu de soluţii, adică un backtracking în care funcţia de continuare lipseşte.

5. Se dă o mulţime $A = \{1, 2, \ldots, n\}$. Să se afişeze toate submulţimile acestei mulţimi.

   Indicaţie: Se generează toţi vectorii caracteristici de lungime n. Prin vector caracteristic se înţelege un vector care are doar valorile 1 sau 0 pentru fiecare element cu semnificaţia:

$$x[i] = \begin{cases} 1 & \textit{dacă } i \textit{ aparţine submulţimii} \\ 0 & \textit{altfel} \end{cases}$$

   Există şi o soluţie care generează vectorii caracteristici de lungime n prin adunarea în baza 2. Iniţial vectorul este nul, corespunzător mulţimii vide, iar apoi prin adunări repetate se generează toate submulţimile. Atenţie, numărul total de submulţimi este 2^n!

6. O firmă dispune de *n* angajaţi, dintre care *p* sunt femei. Firma trebuie să formeze o delegaţie de *m* persoane, dintre care *k* sunt femei. Să se afişeze toate delegaţiile care se pot forma.

   Indicaţie: Pentru a forma o delegaţie de k femei din p disponibile avem la dispoziţie C_p^k variante. Delegaţia de m persoane poate fi completată

cu oricare din variantele de C_{n-p}^{m-k} de alegere a bărbaţilor din delegaţie. Aşadar numărul total de variante este $C_p^k * C_{n-p}^{m-k}$.
Generarea efectivă se bazează pe un vector caracteristic cu semnificaţia:

$$x[i] = \begin{cases} 1 & \text{dacă persoana e femeie} \\ 0 & \text{altfel} \end{cases}$$

Funcţia de continuare va număra femeile din delegaţie şi nu va lăsa ca numărul lor să-l depăşească pe k.

7. Se consideră mulţimea $A = \{1, 2, \ldots, n\}$. Să se furnizeze toate partiţiile acestei mulţimi. (O partiţie a unei mulţimi este o scriere a mulţimii ca reuniune de submulţimi disjuncte).

   Indicaţie: Vom genera partiţia sub forma unui vector cu n componente în care $x[i] = k$ are semnificaţia că elementul i aparţine submulţimii k a partiţiei considerate. Ca exemplu, pentru $n = 4$ putem avea, la un moment dat, vectorul $x = (1, 2, 1, 2)$ ceea ce corespunde partiţiei: $A = \{1, 3\} \cup \{2, 4\}$. Ar fi de remarcat că vectorul caracteristic poate lua valori care vor avea aceeaşi interpretare, ca de exemplu $x = (2, 1, 2, 1)$ ceea ce corespunde partiţiei: $A = \{2, 4\} \cup \{1, 3\}$. Dar reuniunea e comutativă şi partiţia astfel obţinută e identică cu cea anterioară. Pentru a evita acest lucru vom impune ca fiecare componentă a vectorului să poată avea cel mult valoarea k, unde k este indicele elementului. Semnificaţia ar fi că elementul cu indicele 1 va putea face parte doar din submulţimea 1, cel cu indicele 2 doar din submulţimile 1 şi 2 etc. O altă restricţie ar fi aceea că un element nu poate lua o valoarea mai mare ca $max + 1$, unde max este valoare maximă a elementelor de rang inferior. Acest lucru se justifică prin faptul că $x = (1, 1, 3, 1)$ nu ar avea nici o semnificaţie.

8. Un comis-voiajor trebuie să viziteze un număr *n* de oraşe pornind din oraşul numărul 1. El trebuie să viziteze fiecare oraş o singură dată, după care să se întoarcă în oraşul 1. Cunoscând legăturile existente între oraşe, se cere să se găsească toate rutele posibile pe care le poate efectua comis-voiajorul.

   Indicaţie: Se va crea o matrice de adiacenţă (cunoscută din teoria grafurilor), care este o matrice simetrică:

$$A(i,j) = \begin{cases} 1 & \textit{dacă există legătură între oraşele } i \textit{ şi } j \\ 0 & \textit{altfel} \end{cases}$$

Funcţia de continuare va testa dacă la elementul actual se poate ajunge din anteriorul, în vectorul x. Ca observaţie trebuie spus că pentru a obţine soluţiile distincte trebuie făcut un artificiu asemănător cu cel de la problema anterioară.

9. Idem problema anterioară, cu precizarea că pentru fiecare drum între două oraşe se cunoaşte distanţa care trebuie parcursă. Se cere să se găsească ruta de lungime minimă.

   Indicaţie: În momentul reţinerii soluţiei se va calcula lungimea drumului parcurs. Se va compara această lungime cu lungimea anterioară considerată minimă şi se va reţine valoarea actuală minimă împreună cu drumul parcurs.
   Această problemă este celebră prin faptul că este un exemplu pentru imposibilitatea aflării soluţiei exacte în timp polinomial. Datorită complexităţii mari a metodei s-au găsit metode mai puţin complexe (metodele euristice) dar care dau o soluţie cu o marjă de aproximare.

10. Presupunem că avem de plătit o sumă s şi avem la dispoziţie un număr nelimitat de bancnote şi monede de valoare $\nu_1, \nu_2, \ldots, \nu_n$. Să se furnizeze toate variantele de plată a sumei utilizând numai aceste monezi.

11. Idem problema anterioară, cu precizarea că trebuie să plătim suma respectivă cu un număr cât mai mic de monede şi bancnote.

    Indicaţie: Faţă de rezolvarea problemei anterioare se poate face, spre exemplu, o modificare care să compare, în momentul găsirii unei soluţii, numărul de monede cu cel găsit la soluţiile anterioare.

12. Idem problema anterioară pentru cazul în care dispunem doar de un număr $n_1, n_2, \ldots, n_n$ de monede de valoare $\nu_1, \nu_2, \ldots, \nu_n$.

    Indicaţie: Deosebirile faţă de problemele anterioare sunt:
    ** în vectorul soluţie vom reţine numărul de monede sau bacnote folosite, nu şi valoarea lor. Astfel, fiecărui nivel îi corespunde o anumită valoare;*

** vom avea grijă ca fiecare componentă a vectorului soluţie să nu depăşească numărul de monede existent din valoarea care îi corespunde;*
** sumele se vor calcula prin cumularea produselor dintre valoare şi numărul de valori folosite.*

13. Fiind dat un număr natural n, să se genereze toate partiţiile sale. O partiţie a unui număr reprezintă scrierea sa ca sumă de numere naturale nenule.

14. Fiind dat un număr natural n, să se genereze toate descompunerile sale ca sumă de numere prime.

    Indicaţie: Faţă de problema anterioară se poate verifica, la continuare, dacă numărul ales este prim.

15. O fotografie alb-negru este reprezentată sub forma unei matrice cu elemente 0 sau 1. În fotografie sunt reprezentate unul sau mai multe obiecte. Porţiunile corespunzătoare obiectelor au valoarea 1 în matrice. Se cere să se determine dacă fotografia reprezintă unul sau mai multe obiecte.
    Exemplu: Matricea de mai jos reprezintă două obiecte:

$$\begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

16. Un teren dreptunghiular este împărţit în $m \times n$ parcele reprezentate sub forma unei matrice A cu m linii şi n coloane. Fiecare element al matricei este un număr real care reprezintă înălţimea parcelei respective. Pe una dintre parcele se află plasată o bilă. Se cere să se furnizeze toate posibilităţile prin care bila poate să părăsească terenul, cunoscut fiind faptul că bila se poate rostogoli numai pe parcele învecinate a căror înălţime este strict inferioară înălţimii parcelei pe care bila se află.

    Indicaţie: Aceasta şi problema anterioară sunt cazuri tipice de backtracking în plan. Ideea rezolvării constă în încercarea de a ajunge la o poziţie vecină respectând condiţiile problemei. Modalitatea de mişcare este dată de cele 8 direcţii cardinale N, NV, V, SV, S, SE, E, NE. Practic, vectorul soluţie va conţine direcţia în care s-a făcut deplasarea.

17. Pe o tabla de şah de dimensiune 8×8 să se poziţioneze n pioni după următoarele reguli:

    (a) Pe fiecare linie se află doi pioni;

    (b) Pe fiecare coloană se află cel mult doi pioni;

    (c) Pe fiecare paralelă la diagonala principală se află cel mult doi pioni.

12. Divide et impera

Vom găsi o cale, iar dacă nu există,
vom crea una.

Hanibal

În acest capitol vom studia o altă metodă fundamentală de elaborare a algoritmilor, numită *divide et impera*. Ca şi *backtracking*, *divide et impera* se bazează pe un principiu extrem de simplu: *descompunem problema în două (sau mai multe) subprobleme de dimensiuni mai mici, rezolvăm subproblemele, iar soluţia pentru problema iniţială se obţine combinând soluţiile subproblemelor în care a fost descompusă.* Rezolvarea subproblemelor se face în acelaşi mod cu problema iniţială. Procedeul se reia până când subproblemele devin atât de simple încât admit o rezolvare imediată.

Încă din descrierea globală a acestei tehnici s-au strecurat elemente de *recursivitate*. Pentru a putea înţelege mai bine această metodă de elaborare a algoritmilor care este eminamente recursivă, vom prezenta pentru început câteva elemente fundamentale referitoare la recursivitate. Continuăm apoi cu prezentarea generală a metodei, urmată de rezolvarea anumitor probleme de *Divide et Impera* deosebit de importante: *căutare binară, sortarea prin interclasare, sortarea rapidă* şi *evaluarea expresiilor aritmetice.*

În cadrul acestui capitol vom prezenta:

- Ce este recursivitatea şi care este mecanismul prin care ea funcţionează;
- Când se poate aplica metoda *divide et impera* şi care este forma ei generală;
- Cum se aplică această metodă pentru a rezolva eficient problema ordonării.

12.1 Introducere în recursivitate

În acest paragraf vom reaminti câteva elemente esenţiale referitoare la recursivitate. Cei care stăpânesc deja acest mecanism, pot trece direct la prezentarea metodei *Divide et Impera* din paragraful următor.

Având în vedere faptul că recursivitatea este un mecanism de programare general, care nu ţine doar de limbajul Java, prezentarea făcută va folosi şi limbajul pseudocod la descrierea algoritmilor recursivi, pentru a nu încărca prezentarea cu detalii de implementare.

12.1.1 Funcţii recursive

Recursivitatea este un concept care derivă în mod direct din noţiunea de *recurenţă matematică*. Recursivitatea este un instrument elegant şi puternic pe care programatorii îl au la dispoziţie pentru a descrie algoritmii. Este interesant de reţinut faptul că programatorii obişnuiau să utilizeze recursivitatea pentru a descrie algoritmii, cu mult înainte ca limbajele de programare să suporte implementarea directă a acestui concept.

Din punct de vedere informatic, o subrutină (procedură[1] sau funcţie) recursivă este o *subrutină care se autoapelează*.

Să luăm ca exemplu funcţia *factorial*, a cărei definiţie matematică recurentă este:

$$fact(n) = \begin{cases} n * fact(n-1) & \text{pentru} \quad n \geq 1 \\ 1 & \text{pentru} \quad n = 0 \end{cases}$$

Din exemplul de mai sus se observă că factorialul este definit funcţie de el însuşi, dar pentru o valoare a parametrului mai mică cu o unitate. Iată acum care este implementarea recursivă a factorialului, folosind o funcţie algoritmică (stânga) şi implementarea Java (dreapta):

```
funcţie fact(n)
        dacă n = 0 atunci
                fact ← 1
        altfel
                fact ← n*fact(n-1)
return
```

```
public static long fact(int n)
{
        if (n == 0)
                return 1;
        else
                return n*fact(n-1);
}
```

[1] În algoritmică, prin procedură se înţelege o funcţie care nu returnează nici o valuare (de exemplu, în Java o metodă care returnează `void`)

Se observă că funcţia de mai sus nu este decât o "traducere" aproape directă a formulei matematice anterioare. Trebuie să remarcăm că, aşa cum vom vedea în continuarea acestui paragraf, la baza funcţionării acestor funcţii stă un mecanism foarte precis, care nu este atât de trivial cum ar părea la prima vedere.

Să luăm ca al doilea exemplu, calculul celebrului *şir al lui Fibonacci*, care este definit recurent astfel:

$$fib(n) = \begin{cases} fib(n-1) + fib(n-2) & \text{pentru} \quad n > 1 \\ n & \text{pentru} \quad n = 0, 1 \end{cases}$$

Implementarea în pseudocod, respectiv Java, a calculului şirului lui Fibonacci este:

```
funcţie fib(n)
    dacă n=0 sau n=1 atunci
        fib ← n
    altfel
        fib ← fib(n-1)+fib(n-2)
return
```

```
public static long fib(int n)
{
    if (n==0||n==1)
        return n;
    else
        return fib(n-1)+fib(n-2);
}
```

Se observă că în ambele exemple am început cu aşa numita condiţie de terminare:

<u>dacă</u> n = 0 sau n = 1 <u>atunci</u>
fib←n

care corespunde cazului în care nu se mai fac apeluri recursive. O funcţie recursivă care nu are condiţie de terminare va genera *apeluri recursive interminabile*, care se soldează în Java cu o eroare de tipul `StackOverflowError` (*depăşire de stivă*, deoarece aşa cum vom vedea, fiecare apel recursiv presupune salvarea anumitor date pe stivă, iar stiva are o dimensiune finită). Condiţia de terminare ne asigură de faptul că atunci când parametrul funcţiei devine suficient de mic, nu se mai realizează apeluri recursive şi funcţia este calculată direct.

Ideea fundamentală care stă la baza înţelegerii profunde a mecanismului recursivităţii este aceea că *în esenţă, un apel recursiv nu diferă cu nimic de un apel de funcţie obişnuit*. Pentru a veni în sprijinul acestei afirmaţii trebuie să studiem mai în amănunţime ce se petrece în cazul unui apel de funcţie.

Se cunoaşte faptul că în situaţia în care compilatorul întâlneşte un apel de funcţie, acesta predă controlul execuţiei funcţiei respective, după care se revine

la următoarea instrucţiune de după apel. Întrebările care apar în mod firesc sunt: *de unde ştie compilatorul unde să se întoarcă la terminarea funcţiei? De unde ştie care au fost valorile variabilelor înainte de a se preda controlul funcţiei?* Răspunsul este simplu: înainte de a realiza un apel de funcţie, compilatorul salvează complet starea programului (linia de la care s-a realizat apelul, valorile variabilelor locale, valorile parametrilor de apel) pe stivă, urmând ca la revenirea din subrutină să reîncarce de pe stivă starea care a fost înainte de apel.

Pentru exemplificare să considerăm următoarea procedură (nerecursivă) care afişează o linie a unei matrice. Atât linia care trebuie afişată, cât şi matricea sunt transmise ca parametru:

```
procedură AfisLin(a: tmatrice; n, lin: integer)
        pentru i = 1, n
                scrie a[lin, i]
return
```

Procedura *AfisLin* este apelată de procedura *AfisMat* descrisă mai jos, care afişează, linie cu linie, o întreagă matrice pe care o primeşte ca parametru:

```
procedură AfisMat(a: tmatrice; n: integer)
        pentru i = 1, n
                AfisLin(a, n, i)
return
```

Să presupunem că procedura *AfisMat* este apelată într-un program astfel:

```
...
AfisMat(a, 5)
...
```

pentru a afişa o matrice de dimensiuni 5×5.

În momentul în care compilatorul întâlneşte acest apel, el salvează pe stivă linia de la care s-a făcut apelul (să spunem 2181), valoarea matricei *a* şi alte variabile locale declarate în program:

$2181; AfisMat(a, n); \ldots$

Controlul va fi apoi preluat de către procedura *AfisMat*, care intră în ciclul *pentru* cu apelul: *AfisLin(a,n,i)* aflat, să zicem, la linia 2198.

În acest moment controlul va fi preluat de către procedura *AfisLin*, dar nu înainte de a adăuga la vârful stivei linia de la care s-a făcut apelul, valorile parametrilor şi a variabilei locale *i*:

$2198; AfisLin(n, a, i); i = 1; \ldots$
$2181; AfisMat(n, a); \ldots$

Procedura *AfisLin* va tipări prima linie a matricei, după care execuţia ei se încheie. În acest moment compilatorul consultă vârful stivei pentru a vedea unde trebuie să revină şi care au fost valorile parametrilor şi variabilelor locale înainte de apel. Variabila *i* devine 2 şi din nou se apelează procedura *AfisLin*, etc.

Remarcăm aici faptul că atât procedura *AfisMat* cât şi procedura *AfisLin* utilizează o variabilă locală numită *i*. Nu poate exista nici o confuzie între cele două variabile, deoarece în momentul execuţiei lui *AfisLin*, valoarea variabilei *i* din *AfisMat* este salvată pe stivă.

Să vedem acum evoluţia stivei program în cazul calculului recursiv al lui *fact(5)*. Presupunem că la linia 2145 are loc apelul recursiv: $fact \leftarrow n * fact(n-1)$.

Pentru a realiza înmulţirea respectivă, trebuie ca întâi să se calculeze *fact(n-1)*. Cum *n* are valoarea 5, pe stivă se va depune $fact(4)$. Abia după ce valoarea lui $fact(4)$ va fi calculată se poate calcula valoarea lui $fact(5)$. Calculul lui $fact(4)$ implică însă calculul lui $fact(3)$, care implică la rândul lui calculul lui $fact(2)$, $fact(1)$, $fact(0)$. Calculul lui $fact(0)$ se realizează prin atribuire directă, fără nici un apel recursiv:

<u>*dacă*</u> *n=0* <u>*atunci*</u>

$fact \leftarrow 1$

în acest moment, stiva programului conţine toate apelurile recursive realizate până acum:

$2145; fact(0);$
$2145; fact(1);$
$2145; fact(2);$
$2145; fact(3);$
$2145; fact(4);$
$xxxx; fact(5);$

$fact(1)$ fiind calculat, se poate reveni la calculul înmulţirii $2*fact(1) = 2$, apoi, $fact(2)$ fiind calculat se revine la calculul înmulţirii $3*fact(2)$=6 etc., până se calculează $5*fact(4)$=120 şi se revine în programul apelant.

Să vedem acum modul în care se realizează calculul recursiv al şirului lui *Fibonacci*. Vom vedea că timpul de calcul al acestei recurenţe este incomparabil mai mare faţă de calculul factorialului. Să presupunem că funcţia *fib* se apelează cu parametrul $n = 3$. În această situaţie, se depune pe stivă apelul $fib(3)$ împreună cu linia de unde s-a realizat apelul (de exemplu, 2160). În linia 2160 a procedurii are loc apelul recursiv: $fib \leftarrow fib(n-1) + fib(n-2)$ care în cazul nostru, *n* fiind 3, presupune calcularea sumei $fib(2) + fib(1)$. Această sumă nu poate fi calculată înainte de a-l calcula pe $fib(2)$. Calculul lui $fib(2)$ presupune calcularea sumei $fib(1) + fib(0)$. $fib(1)$ şi $fib(0)$ se calculează direct la următorul apel recursiv, după care se calculează suma lor, rezultând că $fib(2) = 2$. Abia acum se revine la suma $fib(2) + fib(1)$ şi se calculează $fib(1)$, după care se revine şi se calculează $fib(3)$.

Modul de calcul al lui $fib(n)$ recursiv se poate reprezenta foarte sugestiv arborescent. Rădăcina arborelui este $fib(n)$, iar cei doi fii sunt apelurile recursive pe care $fib(n)$ le generează, şi anume $fib(n-1)$ şi $fib(n-2)$. Apoi se reprezintă apelurile recursive generate de $fib(n-2)$ ca în **Figura** 12.1.

Din **Figura** 12.1 se observă că anumite valori ale şirului lui *Fibonacci* se calculează (inutil) de mai multe ori. $fib(n)$ şi $fib(n-1)$ se calculează o dată, $fib(n-2)$ se calculează de două ori, $fib(n-3)$ de 3 ori etc. Aceasta explică de ce în capitolul 9 am obţinut o complexitate exponenţială pentru varianta recursivă de calcul a şirului lui Fibonacci.

Figura 12.1: Reprezentarea arborescentă a apelurilor recursive realizate pentru a calcula şirul lui Fibonacci. Se observă că `Fib(n-3)` este calculat de 3 ori.

12.1.2 Recursivitatea nu înseamnă recurenţă

Implementarea recursivă a funcţiilor recurente este uşor de înţeles, datorită mecanismului simplu de transpunere a recurenţei într-o funcţie recursivă. Totuşi, recursivitatea nu se limitează doar la implementarea recurenţelor matematice. Putem defini la fel de bine şi *operaţii* recursive. O operaţie recursivă este definită funcţie de ea însăşi, dar pentru o problemă de dimensiune cu o unitate mai mică. De exemplu, operaţia de inversare a n caractere se poate defini recursiv astfel: se extrage primul caracter din şir, apoi se inversează cele $n - 1$ caractere rămase după care se adaugă la final caracterul extras. Acest principiu îl aplicăm în exemplul care urmează.

Exemplu: Să se scrie o funcţie care citeşte o secvenţă de caractere până când întâlneşte caracterul ".", după care afişează caracterele în ordine inversă.

Rezolvarea acestei probleme se poate formula recursiv astfel: inversarea caracterelor unui şir de n elemente implică inversarea caracterelor rămase după citirea primului caracter, şi scrierea în final a primului caracter:

$$Inv(n) = Citeste(a) + Inv(n-1) + Scrie(a)$$

În consecinţă, metoda de inversare va avea forma (parametrul n a fost eliminat, el fiind dat în formulă doar pentru claritate):

```
funcţie inversare
    citeşte a
    dacă a <>'.' atunci inversare
    scrie a
return
```

```
public static void inversare()
{
    char a;
    a=Reader.readChar();
    if (a!='.')
    {
        inversare();
    }
    System.out.print(a);
}
```

Reamintim că `Reader` este o clasă ajutătoare pentru citirea de date de la tastatură şi a fost definită în cadrul primului volum, fiind reluată şi în cadrul acestui volum, la paragraful 11.4.1.

Este important de notat că pentru ca metoda să funcţioneze corect, variabila *a* trebuie declarată ca variabilă locală; astfel, toate valorile citite vor fi salvate pe stivă, de unde vor fi extrase succesiv (în ordinea inversă citirii) după întâlnirea caracterului ".".

Exemplu: Transformarea unui număr din baza 10 într-o bază *b*, mai mică decât 10.

Să ne reamintim algoritmul clasic de trecere din baza 10 în baza *b*. Numărul se împarte la *b* şi se reţine restul. Câtul se împarte din nou la *b* şi se reţine restul şi se continuă acest procedeu până când câtul devine mai mic decât *b*. Rezultatul se obţine prin scrierea în ordine inversă a resturilor obţinute.

Formularea recursivă a acestei rezolvări pentru trecerea unui număr *n* în baza *b* este:

$$Transf(n) = \begin{cases} Transf(n\,div\,b) + Scrie(n\,mod\,b) & \text{pentru} \quad n \geq b \\ - & \text{pentru} \quad n < b \end{cases}$$

Transpunerea recursivă a formulei anterioare este:

```
funcţie transform(n:integer)
  rest = n mod b
  dacă n ≥ b atunci transform(ndivb)
  scrie rest
return
```

```
public static void transform(int n)
{
  int rest=n%b;
  if(n >= b)
  {
    transform(n/b);
  }
  System.out.print(rest);
}
```

De remarcat că în această funcţie variabila *rest* trebuie să fie declarată local, pentru a fi salvată pe stivă, în timp ce variabila *b* este bine să fie declarată global, deoarece valoarea ei nu se modifică, salvarea ei pe stivă ocupând spaţiu inutil.

Odată ce am înţeles mecanismul recursivităţii, suntem pregătiţi pentru a înţelege cea de-a doua metodă de elaborare a algoritmilor, *Divide et Impera*.

12.2 Prezentarea metodei Divide et Impera

Divide et Impera este o metodă specială prin care se pot aborda anumite categorii de probleme. Ca şi celelalte metode de elaborare a algoritmilor, *Divide et Impera* se bazează pe un principiu extrem se simplu: *se descompune problema iniţială în două (sau mai multe) subprobleme de dimensiune mai mică, după care soluţia problemei iniţiale se obţine combinând soluţiile subproblemelor în care a fost descompusă.* Procedeul de descompunere se repetă până când, după descompuneri succesive, se ajunge la probleme de dimensiune mică, pentru care există rezolvare directă.

Evident, nu orice gen de problemă se pretează la a fi abordată cu *Divide et Impera*. Din descrierea de mai sus reiese că o problemă abordabilă cu această metodă trebuie să aibă două proprietăţi:

1. Să se poată descompune în subprobleme;

2. Soluţia problemei iniţiale să se poată construi simplu pe baza soluţiei subproblemelor.

Modul în care metoda a fost descrisă, conduce în mod natural la o implementare recursivă, având în vedere faptul că şi subproblemele se rezolvă în acelaşi mod cu problema iniţială. Iată care este forma generală a unei funcţii *Divide et Impera*:

```
funcţie DivImp(P: Problemă)
  dacă Simplu(P) atunci
   RezolvăDirect(P);
  altfel
   Descompune(P, P1, P2);
   DivImp(P1);
   DivImp(P2);
   Combină(P1, P2);
return
```

În consecinţă, putem spune că abordarea *Divide et Impera* implică trei paşi la fiecare nivel de recursivitate:

1. *Divide* problema în două subprobleme;

2. *Impera (Stăpâneşte, Cucereşte)* cele două subprobleme prin rezolvarea acestora în mod recursiv;

3. *Combină* soluţiile celor două subprobleme în soluţia finală pentru problema iniţială.

12.3 Căutare binară

Căutarea binară este o metodă eficientă de regăsire a unor valori într-o secvenţă ordonată. Căutarea binară este cel mai simplu exemplu de problemă *Divide et Impera,* deoarece în cazul ei se rezolvă doar una din cele două subprobleme, deci faza de recombinare a soluţiilor nu mai este necesară. Enunţul problemei de căutare binară este:

Se dă un vector cu n componente (întregi), ordonate crescător şi un număr întreg oarecare p. Să se decidă dacă acest număr se găseşte în vectorul dat, şi în caz afirmativ să se furnizeze indicele poziţiei pe care se găseşte.

O rezolvare imediată a problemei presupune parcurgerea secvenţială a vectorului dat, până când p este găsit, sau am ajuns la sfârşitul vectorului. Această rezolvare însă nu foloseşte faptul că vectorul este sortat.

Căutarea binară procedează în felul următor: se compară p cu elementul din mijlocul vectorului, dacă p este egal cu acel element, căutarea s-a încheiat. Dacă este mai mic, se caută doar în prima jumătate, iar dacă este mai mare, se caută doar în a doua jumătate.

Se observă că în această situaţie problema nu se descompune în două subprobleme care se rezolvă, după care se construieşte soluţia, ci se reduce la una sau la alta din subprobleme. Cei trei paşi ai lui Divide et Impera sunt în această situaţie:

1. *Divide:* împarte şirul de n elemente în care se realizează căutarea în două şiruri cu $n/2$ elemente;

2. *Stăpâneşte*: Caută într-una dintre cele două jumătăţi, funcţie de valoarea elementului din mijloc;

3. *Combină*: Nu există.

Metoda binarySearch() din **Listing 12.1** realizează căutarea elementului `el` în şirul `s`, între indicii `low` şi `high`.

Listing 12.1: Implementarea căutării binare. Metoda va returna poziţia pe care se găseşte numărul `el` în şirul `s` sau -1 dacă `el` nu este găsit.

```java
public static int binarySearch(int[] s, int el, int low,
                               int high)
{
    if (low <= high) //conditie de oprire
    {
        int middle = (low + high) / 2;
        if (el == s[middle])
        {
            return middle;
        }
        else
        {
            if (el < s[middle])
            {
                return binarySearch(s, el, low, middle - 1);
            }
            else
            {
                return binarySearch(s, el, middle + 1, high);
            }
        }
    }
    return -1;
}
```

Poziţia pe care se găseşte elementul `el` în şirul `s` este obţinută prin apelul:

```java
poz = binarySearch(s, el, 0, s.length - 1)
```

Listing 12.2 prezintă o clasă simplă care utilizează metoda căutării binare pentru a găsi un număr într-un şir citit de la tastatură:

Listing 12.2: Rezolvarea problemei căutării binare

```java
import java.io.*;
import io.Reader;

/**
 * Program ce verifica daca un element introdus
 * de la tastatura se gaseste in cadrul unui sir ordonat.
 */
public class BinarySearch
{
  /** Metoda de cautare a elementului el in sirul s.*/
  public static int search(int[] s, int el, int low,
```

```
            int high)
{
    if (low <= high)
    {
        int mid = (low + high) / 2;

        if (el == s[mid])
        {
            //element gasit
            return mid;
        }
        else
        {
            if (el < s[mid])
            {
                //cauta in prima jumatate a subsirului
                return search(s, el, low, mid - 1);
            }
            else
            {
                //cauta in a doua jumatate a subsirului
                return search(s, el, mid + 1, high);
            }
        }
    }

    return -1;
}

/** Programul principal.*/
public static void main(String[] args)
{
    //citirea elementelor sirului
    System.out.println("Introduceti elementele sirului in " +
        "ordine crescatoare (pe aceeasi linie):");
    int[] s = Reader.readIntArray();

    //citirea elementului cautat
    System.out.print("Introduceti elementul cautat: ");
    int el = Reader.readInt();

    //cautarea elementului
    int poz = search(s, el, 0, s.length - 1);

    //afisarea rezultatului cautarii
    if (poz > -1)
    {
        System.out.println("Elementul " + el +
            " a fost gasit in sir pe pozitia " +
            poz);
```

```
    }
    else
    {
      System.out.println("Elementul " + el +
        " nu a fost gasit in sir");
    }
  }
}
```

12.4 Sortarea prin interclasare (MergeSort)

Sortarea prin interclasare este, alături de *sortarea rapidă (QuickSort)* şi *sortarea cu ansamble (HeapSort)*, una dintre metodele eficiente de ordonare a elementelor unui şir. Enunţul problemei este următorul:

Să se ordoneze crescător un şir cu n componente întregi.

Principiul de rezolvare constă în a împărţi şirul care trebuie ordonat în două părţi egale şi a ordona fiecare jumătate, după care se interclasează cele două jumătăţi. Descompunerea în două jumătăţi se realizează până când se ajunge la şiruri cu un singur element, care nu mai necesită sortare. Algoritmul de sortare prin interclasare urmează îndeaproape conceptul *Divide et Impera*. Pe scurt, modul lui de operare este următorul:

1. *Divide*: împarte şirul de *n* elemente care urmează a fi sortat în două şiruri cu $n/2$ elemente;
2. *Stăpâneşte*: Sortează recursiv cele două subşiruri utilizând sortarea prin interclasare;
3. *Combină*: Interclasează subşirurile sortate pentru a obţine rezultatul final.

Metoda `mergeSort()` din **Listing 12.3** implementează algoritmul de sortare prin interclasare. Apelul iniţial al funcţiei este:

```
mergeSort(s, 0, s.length - 1);
```

Listing 12.3: Metodă care ordonează şirul `s` folosind sortarea prin interclasare

```
public static void mergeSort(int[] s, int low, int high)
{
    if (low < high)
    {
        int mid = (low + high) / 2;
        mergeSort(s, low, mid);
        mergeSort(s, mid + 1, high);
```

```
            intercls(s, low, mid, high);
        }
}
```

Metoda de interclasare în acest caz este analoagă cu metoda de interclasare obişnuită a două şiruri, diferenţa constând în faptul că acum se interclasează două jumătăţi ale aceluiaşi şir, iar rezultatul se va depune în final tot în şirul interclasat. **Listing 12.4** prezintă implementarea completă a sortării prin interclasare, aplicată pe un şir care este preluat de la tastatură.

Listing 12.4: Soluţia algoritmului de sortare Mergesort

```
import java.io.*;
import io.Reader;

/**
 * Program ce ordoneaza un sir de numere intregi
 * folosind metoda MergeSort.
 */
public class MergeSort
{
  /** Metoda de interclasare a celor 2 subsiruri.*/
  public static void intercls(int low, int mid,
        int high, int[] s)
  {
    int i = low;
    int j = mid + 1;

    int[] inter = new int[high + 1];
    int k = low;

    //interclasarea elementelor
    while ((i <= mid) && (j <= high))
    {
      if (s[i] <= s[j])
      {
        inter[k++] = s[i++];
      }
      else
      {
        inter[k++] = s[j++];
      }
    }

    //au mai ramas elemente din primul subsir?
    for (int l = i; l <= mid; l++)
    {
      inter[k++] = s[l];
    }

```

```
    //au mai ramas elemente din cel de-al doilea subsir?
    for (int l = j; l <= high; l++)
    {
      inter[k++] = s[l];
    }

    //copierea elementelor din sirul inter in sirul s
    for (i = low; i <= high; i++)
    {
      s[i] = inter[i];
    }
  }

  /** Metoda de sortare.*/
  public static void mergeSort(int[] s, int low, int high)
  {
    if (low < high) //subsirul are cel putin 2 elemente
    {
      int mid = (low + high) / 2;
      mergeSort(s, low, mid);
      mergeSort(s, mid + 1, high);
      intercls(low, mid, high, s);
    }
  }

  /** Programul principal.*/
  public static void main(String[] args)
  {
    //citirea elementelor sirului
    System.out.println("Introduceti elementele sirului " +
      "(pe aceeasi linie):");
    int[] s = Reader.readIntArray();

    //sortarea elementelor sirului prin metoda "MergeSort"
    mergeSort(s, 0, s.length - 1);

    //afisarea rezultatului sortarii
    System.out.println(
      "Sirul ordonat prin metoda MergeSort este:");
    for (int i = 0; i < s.length; i++)
    {
      System.out.print(s[i] + " ");
    }
  }
}
```

12.5 Sortarea rapidă (QuickSort)

Sortarea rapidă este, aşa cum îi spune şi numele, cea mai rapidă metodă de sortare prin comparaţii cunoscută în prezent. Există foarte multe variante ale acestei metode, o parte dintre ele având doar rolul de a micşora timpul de execuţie în cazul cel mai nefavorabil. Vom prezenta aici varianta clasică, despre care veţi remarca cu surprindere că este neaşteptat de simplă. Enunţul problemei este identic cu cel de la sortarea prin interclasare, şi anume:

Să se ordoneze crescător un şir de numere întregi.

Metoda de sortare rapidă prezentată în acest paragraf este, dintr-un anumit punct de vedere, complementara metodei *Mergesort*. Diferenţa dintre cele două metode este dată de faptul că, în timp ce la *Mergesort* mai întîi vectorul se împărţea în două părţi după care se sorta fiecare parte şi apoi se interclasau cele două jumătăţi, la *Quicksort* împărţirea se face în aşa fel încât cele două şiruri să nu mai necesite a fi interclasate după sortare, adică primul şir să conţină doar elemente mai mici (nu neapărat ordonate) decât elementele celui de-al doilea şir. Rezultă de aici că în cazul lui *Quicksort*, etapa de recombinare este trivială, deoarece problema este astfel împărţită în subprobleme încât să nu mai fie necesară interclasarea şirurilor. Etapele lui *Divide et Impera* pot fi descrise în această situaţie astfel:

1. *Divide*: Împarte şirul de *n* elemente care urmează a fi sortat în două şiruri, *astfel încât elementele din primul şir să fie mai mici decât elementele din al doilea şir*;

2. *Stăpâneşte*: Sortează recursiv cele două subşiruri utilizând sortarea rapidă;

3. *Combină*: Şirul sortat este obţinut din concatenarea celor două subşiruri sortate.

Funcţia care realizează împărţirea în subprobleme (astfel încât elementele primului şir să fie mai mici decât elementele celui de-al doilea) se datorează lui C. A. Hoare, care a găsit o metodă de a realiza această împărţire (numită *partiţionare*) în timp liniar.

Metoda de partiţionare rearanjează elementele tabloului în funcţie de primul element, numit *pivot*, astfel încât elementele mai mici decât primul element sunt trecute în stânga lui, iar elementele mai mari decât primul element sunt trecute în dreapta lui. De exemplu, dacă avem vectorul:

$$a = (7, 8, 5, 2, 3),$$

atunci procedura de partiţionare va muta elementele 5, 2 şi 3 în stânga lui 7, iar 8 va fi în dreapta lui. Cum se realizează acest lucru? Şirul este parcurs simultan de doi indici: primul indice, *low*, pleacă de la primul element şi este incrementat succesiv, iar al doilea indice, *high*, porneşte de la ultimul element şi este decrementat succesiv. În situaţia în care $a[low]$ este mai mare decât $a[high]$, elementele se interschimbă. Partiţionarea este încheiată în momentul în care cei doi indici se întâlnesc (devin egali) undeva în interiorul şirului. La fiecare pas al algoritmului, fie se incrementează *low*, fie se decrementează *high*; întotdeauna unul dintre cei doi indici, *low* sau *high,* este poziţionat pe *pivot.* Atunci când *low* indică pivotul, se decrementează *high*, iar atunci când *high* indică pivotul se incrementează *low*. Iată cum funcţionează partiţionarea pe exemplul de mai sus. La început, *low* indică primul element, iar *high* indică ultimul element:

$$\begin{array}{ccccccc} a = (& \mathbf{7}, & 8, & 5, & 2, & \mathbf{3} &) \\ & \uparrow & & & & \uparrow & \\ & low & & & & high & \end{array}$$

Deoarece $a[low] > a[high]$ elementele 7 şi 3 se vor interschimba. După interschimbare, pivotul va fi indicat de *high*, deci *low* va fi incrementat:

$$\begin{array}{ccccccc} a = (& \mathbf{3}, & 8, & 5, & 2, & \mathbf{7} &) \\ & & \uparrow & & & \uparrow & \\ & & low & & & high & \end{array}$$

Din nou avem $a[low] > a[high]$, elementele 8 şi 7 se vor interschimba. După interschimbare, pivotul va fi indicat de *low*, deci *high* va fi decrementat:

$$\begin{array}{ccccccc} a = (& \mathbf{3}, & \mathbf{7}, & 5, & 2, & 8 &) \\ & & \uparrow & & \uparrow & & \\ & & low & & high & & \end{array}$$

Din nou avem $a[low] > a[high]$, elementele 7 şi 2 se vor interschimba. După interschimbare, pivotul va fi indicat de *high*, deci *low* va fi incrementat:

$$\begin{array}{ccccccc} a = (& \mathbf{3}, & 2, & 5, & \mathbf{7}, & 8 &) \\ & & & \uparrow & \uparrow & & \\ & & & low & high & & \end{array}$$

De data aceasta avem $a[low] <= a[high]$, deci *low* va fi incrementat din nou, fără a se realiza interschimbări.

Listing 12.5: Metoda de partiţionare. Elementele mai mici decât pivotul, `a[low]`, vor fi aşezate în stânga lui, iar elementele mai mari în dreapta. Metoda va returna poziţia pe care se află pivotul.

```
public static int partition(int low, int high)
{
    //variabila care ne spune daca high indica pivotul
    boolean pozPivot = false;
    while (low < high) //indicii nu s-au suprapus
    {
        if (a[low] > a[high])
        {
            interschimba(a, low, high);
            //celalalt indice indica acum pivotul
            pozPivot = !pozPivot;
        }
        pozPivot ? low++ : high--;
    }

    return low; //se returneaza pozitia pivotului
}
```

În acest moment *low* şi *high* s-au suprapus (au devenit egale), deci partiţionarea s-a încheiat. Pivotul este pe poziţia a 4-a, *care este de fapt şi poziţia lui finală în şirul sortat.*

Metoda `partition()` din **Listing 12.5** primeşte ca parametri limitele inferioară, respectiv superioară ale şirului care se partiţionează şi returnează poziţia pe care se află pivotul în finalul partiţionării. Poziţia pivotului este importantă deoarece ne dă locul în care şirul va fi despărţit în două subşiruri.

Două observaţii importante merită făcute referitor la metoda `partition()`:

1. Variabila `pozPivot` poate lua valoarea `false` dacă pivotul este indicat de `low`, sau `true` dacă pivotul este indicat de `high`. Atribuirea *pozPivot =!pozPivot* are ca efect schimbarea stării acestei variabile din `false` în `true` sau invers;

2. Metoda se foloseşte în mod inteligent de transmiterea prin valoare a parametrilor, deoarece modifică variabilele `low` şi `high` bazându-se pe faptul că această modificare nu va afecta valorile lui `low` si `high` din metoda `quickSort()`.

Metoda de ordonare propriu-zisă din **Listing 12.6** respectă structura *Divide et Impera* obişnuită, doar că funcţia de recombinare a soluţiilor nu mai este necesară, deoarece am realizat partiţionarea înainte de apel.

Listing 12.6: Metoda de ordonare `quickSort`

```
public static void quickSort(int low, int high)
{
    if (low < high) //subsirul mai are cel putin 2 elemente
    {
        int mid = partitionare(low, high); //partitioneaza
        quickSort(low, mid - 1); //sorteaza prima jumatate
        quickSort(mid + 1, high); //sorteaza a doua jumatate
    }
}
```

Programul din **Listing 12.7** realizează ordonarea folosind *Quicksort* a unui şir citit de la tastatură.

Listing 12.7: Soluţia problemei de ordonare prin metoda quicksort

```
import java.io.*;
import io.Reader;

/**
 * Program ce realizeaza sortarea unui sir de
 * numere intregi prin metoda quicksort.
 */
public class Quicksort
{
  /** Metoda de partitionare a sirului s.*/
  public static int partitionare(int[] s, int low, int high)
  {
    int pozPivot = 0;
    int aux;

    while (low < high)
    {
      if (s[low] > s[high])
      {
        //interschimbarea elementelor s[low] si s[high]
        aux = s[low];
        s[low] = s[high];
        s[high] = aux;

        pozPivot = 1 - pozPivot;
      }

      if (pozPivot == 0)
```

```
            {
                high--;
            }
            else
            {
                low++;
            }
        }

        return low;
    }

    /** Metoda de sortare a sirului s.*/
    public static void sort(int[] s, int low, int high)
    {
        if (low < high) //subsirul are cel putin 2 elemente
        {
            int mid = partitionare(s, low, high);
            sort(s, low, mid - 1);
            sort(s, mid + 1, high);
        }
    }

    /** Programul principal.*/
    public static void main(String[] args)
    {
        //citirea elementelor sirului
        System.out.println("Introduceti elementele sirului " +
            "(pe aceeasi linie):");
        int[] s = Reader.readIntArray();

        //sortarea elementelor sirului prin metoda "quicksort"
        sort(s, 0, s.length - 1);

        //afisarea rezultatului sortarii
        System.out.println(
            "Sirul ordonat prin metoda quicksort este:");
        for (int i = 0; i < s.length; i++)
        {
            System.out.print(s[i] + " ");
        }
    }
}
```

12.6 Expresii aritmetice

Se dă o expresie aritmetică în care operanzii sunt simbolizaţi prin litere mici (de la a la z), iar operatorii sunt '+', '-', '/' şi '' cu semnificaţia cunoscută. Se cere să se scrie un program care transformă expresia în formă poloneză postfixată.*

Reamintim faptul că *forma poloneză postfixată (Lukasiewicz)* este obţinută prin scrierea operatorului după cei doi operanzi, şi nu între ei. Această formă are avantajul că nu necesită paranteze pentru a schimba prioritatea operatorilor. Ea este utilizată adeseori în informatică pentru a evalua expresii. Iată câteva exemple:

1. *a+b se scrie ab+*
2. *a*(b+c) se scrie abc+**
3. *(a+b)*(c+d) se scrie ab+cd+**

Unul dintre cei mai simpli algoritmi de a trece o expresie în formă poloneză constă în a căuta care este operatorul din expresie cu prioritatea cea mai mică, şi de a aşeza acest operator la sfârşitul expresiei, urmând ca prima parte a formei poloneze să fie formată din transformarea expresiei din stânga operatorului, iar a doua parte a formei poloneze să fie formată din transformarea expresiei din dreapta operatorului.

Cele două subexpresii urmează a se trata în mod analog, până când se ajunge la o subexpresie de lungime 1, care va fi obligatoriu un operand şi care nu mai necesită transformare.

Schematic, dacă avem expresia:

$$E = E1\, op\, E2$$

unde $E1$ şi $E2$ sunt subexpresii, iar *op* este operatorul cu prioritatea cea mai mică (deci operatorul unde expresia se poate "rupe" în două), atunci forma poloneză a lui *E*, notată *Pol(E)*, se obţine astfel:

Pol(E)=Pol(E1) Pol(E2) op.

Expresia de mai sus exprimă faptul că forma poloneză postfixată a lui *E* se obţine prin scrierea în formă poloneză postfixată a celor două subexpresii, urmate de operatorul care le separă. Expresia de mai sus este o expresie recursivă specifică tehnicii *Divide et Impera*. Etapele sunt în această situaţie:

1. *Divide*: împarte expresia aritmetică în două subexpresii legate printr-un operator de prioritate minimă;

2. *Stăpâneşte*: Transformă recursiv în formă poloneză cele două subexpresii;

3. *Combină*: Scrie cele două subexpresii în formă poloneză urmate de operatorul care le leagă.

Soluţia problemei expresiilor aritmetice este dată în **Listing 12.8**. Transformarea propriu-zisă este realizată de metoda `polonez()` de la liniile 39-86. `polonez()` primeşte 3 parametri, care descriu subşirul care trebuie trecut în formă poloneză: x, care reprezintă expresia E la care se adaugă `low` şi `high`, care delimitează subşirul din E care va fi procesat.

Ca orice metodă recursivă, `polonez()` începe la linia 43 cu condiţia de terminare:

```
if(low == high)
```

care corespunde cazului în care şirul are un singur caracter (care trebuie în mod obligatoriu să fie un operand) şi a cărui transformare în formă poloneză este banală.

Ciclul `while` de la liniile 49-54 realizează un lucru foarte important: elimină eventualele paranteze inutile care înconjoară expresia, pentru a putea găsi ulterior uşor operatorul de unde se "rupe" expresia în două. De exemplu, expresia $(a + b) * (c + d)$ va fi ruptă într-o primă etapă în $(a + b)$ şi $(c + d)$. Ambele expresii rezultate sunt înconjurate de paranteze care sunt acum inutile, şi care ar împiedica găsirea operatorului cu prioritate minimă (care trebuie să fie în afara oricărei paranteze). Eliminarea parantezelor exterioare se face utilzând metoda `parant()` de la liniile 14-37, care întoarce `true` dacă parantezele exterioare pot fi eliminate şi `false` în caz contrar.

Ciclul `for` de la liniile 56-82 conţine paşii de *divide* şi *combină* ai metodei. Se încearcă mai întâi găsirea unui operatori aditiv ($+$ sau $-$) şi apoi a unui operator multiplicativ ($*$ sau $/$) care să se afle în afara oricărei paranteze. Dacă un astfel de operator este găsit (liniile 73-75), expresia este "ruptă" în acel loc, iar metoda întoarce transformarea în formă poloneză a primei jumătăţi, concatenată cu transformarea celei de-a doua jumătăţi urmate de operatorul unde s-a realizat împărţirea. Căutarea operatorului de prioritate minimă se face de la dreapta la stânga, şi nu de la stânga la dreapta cum era de aşteptat, pentru a transorma corect expresii de forma $a-b-c$ sau $a/b/c$ (de exemplu, dacă am transforma $a-b-c$ de la stânga la dreapta, am obţine forma poloneza $-a-bc$, care se traduce

prin $a - (b - c)$, ceea ce este incorect; transformarea corectă, obţinută prin parcurgerea de la dreapta la stânga este $--abc$). Dacă se ajunge la instrucţiunea `return` de la linia 85, înseamnă că nu s-a găsit nici un operator de la care expresia să se rupă în două, deci expresia este incorectă.

Listing 12.8: Rezolvarea problemei expresiilor aritmetice

```
1 import java.io.*;
2 import io.Reader;
3
4 /**
5  * Program care transforma o expresie aritmetica
6  * in forma poloneza postfixata.
7  */
8 public class Expresii
9 {
10   /**
11    * Verifica daca expresia mai este corecta
12    * dupa eliminarea parantezelor exterioare.
13    */
14   public static boolean parant(String x, int low, int high)
15   {
16     int nrp = 0; //numarul de paranteze deschise si neinchise
17
18     for (int i = low + 1; i < high; i++)
19     {
20       if (x.charAt(i) == '(')
21       {
22         nrp++;
23       }
24
25       if (x.charAt(i) == ')')
26       {
27         nrp--;
28       }
29
30       if (nrp < 0) //s-a inchis o paranteza fara pereche
31       {        //deci nu putem elimina paranetezele exterioare
32         return false;
33       }
34     }
35
36     return true; //parantezele exterioare pot fi eliminate
37   }
38
39   /** Trece o expresie in forma poloneza postfixata.*/
40   public static String polonez(String x, int low,
41           int high, char[][] op)
42   {
43     if (low == high)
```

```
    {
      return x.valueOf(x.charAt(low));
    }

    //elimina parantezele exterioare inutile
    while (x.charAt(low) == '(' && x.charAt(high) == ')'
      && parant(x, low, high))
    {
      low++;
      high--;
    }
    //cauta locul unde sirul poate fi "rupt" in doua
    for (int i = 0; i < op.length; i++)
    {
      int nrp = 0;

      for (int j = high; j >= low; j--)
      {
        if (x.charAt(j) == '(')
        {
          nrp++;
        }

        if (x.charAt(j) == ')')
        {
          nrp--;
        }
        //daca ne aflam in afara parantezelor si am
        //gasit un operator cu prioritatea adecvata
        if (nrp == 0 &&
          (x.charAt(j) == op[i][0] ||
            x.charAt(j) == op[i][1]))
        {
          return polonez(x, low, j - 1, op) +
            polonez(x, j + 1, high, op) +
            x.valueOf(x.charAt(j));
        }
      }
    }
    //daca s-a ajuns aici, sirul nu a putut fi "rupt" in
    //doua, deci expresia este incorecta
    return "Sirul este incorect";
  }

  /** Programul principal.*/
  public static void main(String[] args)
  {
    //matricea celor 4 operatori standard
    char[][] op = { {'+', '-'}, {'*', '/'} };

```

```
    //citirea expresiei aritmetice
    System.out.print("Introduceti expresia aritmetica: ");

    String x = Reader.readString();
    if (x.length() == 0) return;

    System.out.println("Forma poloneza postfixata a " +
      "expresiei aritmetice este: " +
      polonez(x, 0, x.length() - 1, op));
  }
}
```

Rezumat

La începutul acestui capitol am prezentat elemente esenţiale referitoare la recursivitate. Am văzut care este mecanismul care stă la baza acestei tehnici şi faptul că un apel recursiv nu diferă în esenţă cu nimic de un apel obişnuit. Iată care sunt regulile de bază ale recursivităţii, pe care este bine să le reţineţi şi aplicaţi întotdeauna:

1. *Condiţia de terminare:* cel puţin o instanţă a problemei trebuie întotdeauna să se poată rezolva fără a utiliza recursivitatea;

2. *Progresul:* orice apel recursiv trebuie să progreseze către condiţia de terminare;

3. *Crede şi nu cerceta:* întotdeauna trebuie să presupuneţi că apelul recursiv funcţionează (fără a vă pune problema *cum*);

4. *Suprapunere de apeluri:* evitaţi să executaţi acelaşi lucru de două ori, prin rezolvarea aceleiaşi instanţe a problemei în apeluri recursive distince (cum am făcut la şirul lui Fibonacci).

Recursivitatea are multe aplicaţii concrete, câteva dintre ele fiind prezentate chiar în cadrul acestui capitol.

Introducerea elementelor principale ale recursivităţii a fost urmată apoi de prezentarea metodei *divide et impera*, precum şi a celor trei etape fundamentale care o caracterizează: *divide, cucereşte, combină*. Căutarea binară este o metodă de a găsi rapid un element în cadrul unui şir ordonat. *Quicksort* şi *Mergesort* sunt metode deosebit de eficiente de a ordona un şir. Problema expresiilor aritmetice presupune trecerea unei expresii din forma standard în forma poloneză postfixată.

Noţiuni fundamentale

căutare binară: metodă de căutare eficientă a unui element în cadrul unui şir ordonat.

condiţie de terminare: condiţie care indică sfârşitul recursiei prin rezolvarea directă a unei instanţe a problemei.

Mergesort: algoritm de sortare eficient în care şirul se împarte în două jumătăţi, după care se ordonează fiecare jumătate şi se combină rezultatul.

metodă recursivă: o metodă care se autoapelează direct sau indirect.

Quicksort: algoritm de sortare eficient în care şirul se partiţionează în două, după care se ordonează fiecare partiţie.

suprapunere de apeluri: situaţie în care un algoritm recursiv realizează inutil aceleaşi apeluri de mai multe ori, scăzând astfel drastic eficienţa rezolvării.

Erori frecvente

1. Cea mai frecventă eroare la începători este de a uita să stabilească o condiţie de terminare pentru apelurile recursive.

2. Fiţi atenţi ca fiecare apel recursiv să constituie un pas către condiţia de terminare, altfel recursia este incorectă.

3. Trebuie evitată suprapunerea apelurilor recursive, deoarece ele tind să genereze algoritmi de complexitate exponenţială.

4. Complexitatea algoritmilor recursivi trebuie calculată folosind formule de recurenţă. Nu puteţi presupune că un apel recursiv are o complexitate în timp liniară.

5. În cazul unui apel recursiv, doar variabilele locale şi parametri actuali se salvează pe stivă. Nu vă bazaţi pe faptul că variabilele definite în afara funcţiei sunt salvate la apelul recursiv.

6. Pe de altă parte, evitaţi să declaraţi variabile locale sau parametri formali care nu sunt necesari pentru metoda recursivă, pentru a nu umple stiva cu informaţii inutile.

7. Deşi multe probleme admit descompunerea în două sau mai multe subprobleme, nu întotdeauna soluţia problemei iniţiale se poate obţine pe baza soluţiei subproblemelor.

Exerciţii

Teorie

1. Reprezentaţi evoluţia stivei pentru funcţiile recursive din acest capitol.

2. Calculaţi de câte ori se recalculează valoarea F_k în cazul calculului recursiv al valorii F_n a *şirului lui Fibonacci,* prezentat în paragraful 12.1.1.

În practică

1. Să se calculeze recursiv şi iterativ cel mai mare divizor comun a două numere după formulele (*Euclid*):

$$cmmdc(a,b) = \begin{cases} cmmdc(b, a \, mod \, b) & \text{pentru} \quad a \, mod \, b \neq 0 \\ b & \text{altfel} \end{cases}$$

$$cmmdc(a,b) = \begin{cases} cmmdc(b, |a-b|) & \text{pentru} \quad a \neq b \\ b & \text{altfel} \end{cases}$$

2. Să se calculeze recursiv şi iterativ *funcţia lui Ackermann*, dată de formula:

$$Ack(m,n) = \begin{cases} n+1 & \text{pentru} \quad m = 0 \\ Ack(m-1, 1) & \text{pentru} \quad n = 0 \\ Ack(m-1, Ack(m, n-1)) & \text{altfel} \end{cases}$$

3. Să se calculeze combinările după formula de recurenţă din *triunghiul lui Pascal*:

$$C_n^k = C_{n-1}^k + C_{n-1}^{k-1}$$

   Calculaţi apoi combinările după formula clasică:

$$C_n^k = \frac{n!}{k!(n-k)!}$$

   Ce constataţi? Cum explicaţi ceea ce aţi constatat?

4. Să se calculeze recursiv şi iterativ *funcţia Manna-Pnueli*, dată de formula:

$$F(x) = \begin{cases} x - 1 & \text{pentru} \quad x \geq 12 \\ F(F(x+2)) & \text{altfel} \end{cases}$$

5. Să se scrie o funcţie care calculează recursiv suma cifrelor unui număr după formula:

$$S(n) = \begin{cases} n \, mod \, 10 \; + S(n \, div \, 10) & \text{pentru} \quad n > 0 \\ 0 & \text{pentru} \quad n = 0 \end{cases}$$

6. Se consideră două şiruri definite recurent după formulele:
$a_n = \frac{a_{n-1}+b_{n-1}}{2}$ şi $b_n = \sqrt{a_{n-1}b_{n-1}}$, cu $a_0 = a$ şi $b_0 = b$.
Să se scrie un program recursiv care calculează aceste şiruri.

7. (*Partiţiile unui număr*) Un număr natural *n* se poate descompune ca sumă descrescătoare de numere naturale. De exemplu, pentru numărul 4 avem descompunerile 2+1+1 sau 3+1. Prin $P(n,k)$ se notează numărul de posibilităţi de a-l descompune pe *n* ca sumă (descrescătoare) de k numere. De exemplu, $P(4,2) = 2$ $(4 = 3+1, 4 = 2+2)$. Numerele $P(n,k)$ verifică relaţia de recurenţă:

$$P(n+k,k) = P(n,1) + P(n,2) + \ldots + P(n,k)$$

cu $P(n,1) = P(n,n) = 1$.
Să se calculeze numărul total de descompuneri ale numărului *n*.

8. Să se modifice metoda `polonez()` din **Listing 12.8** astfel încât să poată transforma corect şi expresii în care operanzii au lungimi mai mari de un caracter. De exemplu $(max + 1) * (min + 2)$ etc.

9. Să se scrie o funcţie care calculează maximul elementelor unui şir utilizând tehnica Divide et Impera.

   Indicaţie: Se împarte şirul în două jumătăţi egale, se calculează recursiv maximul celor două jumătăţi şi se alege numărul mai mare.

10. (*Turnurile din Hanoi*) Se dau trei tije simbolizate prin literele A, B şi C. Pe tija A se află *n* discuri de diametre diferite aşezate descrescător în ordinea diametrelor, cu diametrul maxim la bază. Se cere să se mute discurile pe tija B respectând următoarele reguli:

(a) la fiecare pas se mută un singur disc;

(b) nu este permisă aşezarea unui disc cu diametru mai mare peste un disc cu diametrul mai mic.

Indicaţie: Formularea recursivă a soluţiei este: se mută primele n-1 discuri de pe tija A pe tija C folosind ca tijă intermediară tija B; se mută discul rămas pe A pe tija B; se mută discurile de pe tija C pe tija B folosind ca tijă intermediară tija A. Parcurgerea celor trei etape permite definirea recursivă a şirului H(n,a,b,c) astfel:

$$H(n,a,b,c) = \begin{cases} ab & \textit{dacă} \quad n=1 \\ H(n-1,a,c,b), ab, H(n-1,c,b,a) & \textit{dacă} \quad n>1 \end{cases}$$

Exemplu: Pentru $n = 2$ avem:

$$H(2,a,b,c) = H(1,a,c,b), ab, H(1,c,b,a) = ac, ab, cb$$

11. Scrieţi un program în care calculatorul să ghicească cît se poate de repede un număr natural la care v-aţi gândit. Numărul este cuprins între 1 şi 32.000. Atunci când calculatorul propune un număr *i* se va răspunde prin 1, dacă numărul este prea mare, 2 dacă numărul este prea mic şi 0 dacă numărul a fost ghicit.

    Indicaţie: Problema foloseşte metoda căutarii binare prezentată în acest capitol.

12. (*Problema tăieturilor*) Se dă o bucată dreptunghiulară de tablă de dimensiune $l \times h$, având pe suprafaţa ei *n* găuri de coordonate numere întregi (colţul din stânga jos al tablei este considerat centrul sistemului de coordonate). Să se determine care este bucata de arie maximă fără găuri care poate fi decupată din suprafaţa originală. Sunt permise doar tăieturi orizontale sau verticale.

    Indicaţie: Se caută în bucata curentă prima gaură. Dacă o astfel de gaură există, atunci problema se împarte în alte patru subprobleme de acelaşi tip. Dacă suprafaţa nu are nici o gaură, atunci se compară suprafaţa ei cu suprafeţele fără gaură obţinute până la acel moment. Dacă suprafaţa este mai mare, atunci se reţin coordonatele ei.

Coordonatele găurilor sunt date în doi vectori xv şi yv. Coordonatele dreptunghiurilor care apar pe parcursul problemei sunt reţinute prin colţul stânga jos (x,y), lungime şi lăţime (l,h).

Pentru a se afla în interioul unui dreptunghi o gaură trebuie să îndeplinească simultan condiţiile:

(a) $xv(i) > x$;
(b) $xv(i) < x + l$;
(c) $yv(i) > y$;
(d) $yv(i) < y + h$.

Tăietura verticală prin această gaură determină două dreptunghiuri:

(a) *x,y,xv(i)-x,h;*
(b) *xv(i),y,l+x-xv(i),h.*

Tăietura orizontală prin această gaură determină alte două dreptunghiuri:

(a) *x,y,l,yv(i)-y;*
(b) *x,yv(i),l,h+y-yv(i).*

13. Algoritmi Greedy

> Am constatat că cu cât muncesc mai mult, cu atât am mai mult noroc.
>
> Thomas Jeferson

Algoritmii aplicaţi problemelor de optimizare (în care se urmăreşte obţinerea minimului sau maximului unei funcţii obiectiv) sunt, în general, compuşi dintr-o secvenţă de paşi, la fiecare pas existând mai multe alegeri posibile. Pentru multe probleme de optimizare, utilizarea metodei programării dinamice (prezentată în capitolul 14) în vederea determinării celei mai bune soluţii se dovedeşte a fi o strategie prea complicată. Un algoritm Greedy va alege la fiecare moment soluţia care pare a fi cea mai bună la momentul respectiv. Este vorba deci despre o alegere optimă, făcută local, cu speranţa că ea va conduce la un optim global. Acest capitol tratează probleme de optimizare care pot fi rezolvate cu ajutorul algoritmilor Greedy. Algoritmii Greedy conduc în multe cazuri la soluţii optime, dar nu întotdeauna... În secţiunea 13.1 vom prezenta mai întâi o problemă simplă dar netrivială, problema selectării activităţilor, a cărei soluţie poate fi calculată în mod eficient cu ajutorul unei metode de tip Greedy. Mai departe, în secţiunea 13.2 se recapitulează câteva elemente de bază ale metodei Greedy. Capitolul se încheie cu prezentarea câtorva probleme specifice.

Metoda Greedy este destul de puternică şi este aplicată cu succes unui spectru larg de probleme. Lucrările despre teoria grafurilor conţin mai mulţi algoritmi care pot fi priviţi ca aplicaţii ale metodei Greedy, cum ar fi algoritmii de determinare a arborelui parţial de cost minim (Kruskal, Prim) sau algoritmul lui Dijkstra pentru determinarea celor mai scurte drumuri pornind dintr-un vârf.

În cadrul acestui capitol vom vedea:

- Care sunt elementele care caracterizează o strategie greedy;

- Ce este substructura optimă (principiul optimalităţii) şi principiul alegerii greedy;
- Cum se poate demonstra corectitudinea unui algoritm.

13.1 Problema spectacolelor (selectarea activităţilor)

Primul exemplu pe care îl vom considera este o problemă de repartizare a unei resurse (o sală de spectacol) mai multor activităţi care concurează pentru a obţine resursa respectivă (diferite spectacole care au loc în sala respectivă). Vom vedea că un algoritm de tip Greedy reprezintă o metodă simplă şi elegantă pentru programarea unui număr maxim de spectacole care nu se suprapun (numite activităţi compatibile reciproc).

Să presupunem că dispunem de o mulţime S = 1,2,...,n de n activităţi (spectacole) care doresc să folosească o aceeaşi resursă (sala de spectacole). Această resursă poate fi folosită de o singură activitate la un moment dat. Fiecare activitate i are un timp de start s_i *şi un timp de terminare* t_i*, unde* $s_i \leq t_i$*. Dacă este selectată activitatea i, ea se desfăşoară pe durata intervalului* $[s_i, t_i)$*. Două activităţi sunt compatibile dacă duratele lor de desfăşurare sunt disjuncte. Problema spectacolelor (selectării activităţilor) constă în selectarea unei mulţimi maximale de activităţi compatibile între ele.*

Un algoritm Greedy pentru această problemă este descris de următoarea funcţie, prezentată în pseudocod. Vom presupune că spectacolele (adică datele de intrare) sunt ordonate crescător după *timpul de terminare:*

$$t_1 \leq t_2 \leq, \ldots, \leq t_n.$$

În cazul în care activităţile nu sunt ordonate astfel, ordonarea poate fi făcută în timpul $O(n * logn)$ (folosind *Mergesort* sau *Quicksort* prezentate în capitolul anterior). Algoritmul de mai jos presupune că datele de intrare *s* şi *t* sunt reprezentate ca şiruri.

```
functie SELECT-SPECTACOLE-GREEDY(s, t)
  //SS = mulţimea spectacolelor selectate
  SS ← {1}
  //uss = indicele Ultimului Spectacol
  //Selectat
  uss ← 1
```

```
  pentru sc = 2,n //sc este spectacolul curent
    dacă s_sc ≥ t_uss atunci
      //spectacolul curent începe
      //după ce uss s-a terminat, deci
      //se adaugă sc la spect. selectate
      SS ← SS ∪ {sc}
      //ultimul spectacol selectat devine sc
      uss ← sc
return SS
```

În mulţimea SS se introduc spectacolele care au fost selectate. Variabila *uss* identifică ultimul spectacol introdus în SS. Deoarece activităţile sunt considerate în ordinea crescătoare a timpilor lor de terminare, t_{uss} va reprezenta întotdeauna timpul maxim de terminare a oricărei activităţi din SS. Aceasta înseamnă că:

$$t_{uss} = max\{t_k | k \in SS\}$$

La începutul algoritmului, mulţimea spectacolelor selectate, SS, se iniţializează cu activitatea 1 (deoarece această activitate se termină cel mai repede), iar variabila *uss* (ultimul spectacol selectat) ia ca valoare această activitate. În continuare, în ciclul *pentru* se consideră pe rând fiecare activitate, *sc*. Aceasta se adaugă mulţimii SS dacă este compatibilă cu celelalte activităţi deja selectate. Pentru a vedea dacă specacolul curent, *sc*, este compatibil cu toate celelalte activităţi existente la momentul curent în SS, este suficient ca momentul de start s_{sc} să nu fie mai devreme decât momentul de terminare t_{uss} al activităţii cel mai recent adăugate mulţimii SS. Dacă specacolul curent, *sc*, este compatibil, atunci el este adăugat mulţimii SS, iar variabila *uss* este actualizată. Funcţia *SELECT-SPECTACOLE-GREEDY* este foarte eficientă. Ea poate planifica o mulţime *S* de *n* activităţi în $O(n)$, presupunând că activităţile au fost deja ordonate după timpul lor de terminare. Activitatea aleasă de *SELECT-SPECTACOLE-GREEDY* este întotdeauna cea cu primul timp de terminare care poate fi planificată fără suprapuneri.

13.1.1 Demonstrarea corectitudinii algoritmului

Până în acest moment, nu ne-am pus problema demonstrării corectitudinii algoritmilor prezentaţi. Totuşi, trebuie să menţionăm că există o întreagă ramură a algoritmicii care se ocupă exclusiv de demonstrarea corectitudinii algoritmilor. În general, demonstrarea corectitudinii unui algoritm este un proces

destul de laborios, şi, în majoritatea aplicaţiilor practice concrete, este mai adecvată testarea comportamentului pe mulţimi de date reprezentative decât demonstrarea riguroasă a corectitudinii. Există totuşi situaţii în care trebuie să ne asigurăm că o anumită secvenţă critică din cadrul unui program face *exact* ceea ce intenţionăm, indiferent de setul de date care este primit la intrare. De exemplu, pentru un sistem de operare, este esenţial ca algoritmul de planificare al proceselor să funcţioneze corect, indiferent de numărul şi natura proceselor care trebuie planificate. Oricâte seturi de date am alege pentru a testa algoritmul, rămâne posibilitatea ca să fi scăpat din vedere o anumită configuraţie pentru care algoritmul ar funcţiona greşit. Astfel, singura posibilitate de a ne convinge că algoritmul va funcţiona corect indiferent de setul de date de la intrare este să *demonstrăm* riguros corectitudinea lui.

Pentru algoritmul de faţă am ales să îi demonstrăm corectitudinea, deoarece pe de o parte ea este ilustrativă pentru o întreagă clasă de probleme, iar pe de altă parte demonstraţia nu este dificilă.

Teorema 13.1.1 *Algoritmul SELECT-SPECTACOLE-GREEDY furnizează soluţia optimă (număr maxim de spectacole) pentru problema selectării activităţilor.*

Demonstraţie: Fie $S = \{1, 2, \ldots, n\}$ mulţimea activităţilor care trebuie planificate, ordonate crescător după timpul de terminare. În consecinţă, activitatea 1 se termină cel mai devreme. Vom arăta că există o soluţie optimă care începe cu activitatea 1.

Să presupunem că avem o soluţie $A \subseteq S$ optimă pentru o instanţă a problemei. Pentru simplitate, presupunem că activităţile din *A* sunt ordonate după timpul de terminare. Dacă primul spectacol din *A* este chiar 1, atunci demonstraţia este încheiată. Dacă primul spectacol din *A* nu este 1, atunci înlocuim primul spectacol cu spectacolul 1, obţinând evident o soluţie corectă, deoarece spectacolul 1 se va termina mai devreme decât primul spectacol din *A*. Am arătat astfel că există o soluţie optimă pentru *S* care începe cu activitatea 1.

Mai mult, odată ce este făcută alegerea activităţii 1, problema se reduce la determinarea soluţiei optime pentru activităţile din *S* care sunt compatibile cu activitatea 1. Fie $S' = \{i \in S | s_i \geq t_1\}$ mulţimea activităţilor care încep după ce 1 se termină. Rezultă că dacă *A* este o soluţie optimă pentru *S*, atunci $A' = A - \{1\}$ este o soluţie optimă pentru *S'*. Dacă nu ar fi aşa, atunci ar exista o soluţie optimă *B'* pentru *S'* care să aibă mai multe activităţi decât *A'*. Adăugând activitatea 1 la *B'*, vom obţine o soluţie pentru *S* cu mai multe activităţi decât soluţia *A*, ceea ce este absurd.

Astfel, prin inducţie după numărul de alegeri făcute se poate arăta că alegând primul spectacol compatibil la fiecare pas, se obţine o soluţie optimă.

13.1.2 Soluţia problemei spectacolelor

Soluţia problemei spectacolelor este dată în **Listing 13.1**. Metoda `select-Spectacole()` de la liniile 12-33 este transpunerea în Java a funcţiei *SELECT-SPECTACOLE-GREEDY*. `selectSpectacole()` primeşte ca parametri două şiruri reprezentând timpul de început respectiv timpul de sfârşit al fiecărui spectacol şi întoarce un vector cu specacolele care au fost planficate, în ordinea crescătoare a timpului de începere. Ordonarea crescătoare a spectacolelor este realizată de metoda `ordonare()` de la liniile 35-70, care, pentru simplitate, foloseşte metoda bulelor. De remarcat faptul că metoda `ordonare()` trebuie să interschimbe şi valorile din şirul `s`, pentru a menţine consistenţa cu şirul `t`.

Listing 13.1: Soluţia problemei spectacolelor

```
1 import java.util.*;
2 import java.io.*;
3 import io.Reader;
4
5 /**
6  * Problema spectacolelor (selectarea activitatilor prin
7  * metoda Greedy)
8  */
9 public class Spectacole
10 {
11   /** Selectarea spectacolelor. */
12   public static Vector selectSpectacole(int[] s, int[] t)
13   {
14     Vector sol = new Vector();
15
16     if (s.length == 0) return sol;
17
18     //primul spectacol face parte din solutie
19     sol.addElement(new Integer(0));
20
21     int j = 0;
22
23     for (int i = 1; i < s.length; i++)
24     {
25       if (s[i] >= t[j])
26       {
27         sol.addElement(new Integer(i));
28         j = i;
29       }
30     }
31
32     return sol;
33   }
34
35   /** Ordonarea timpilor de terminare a spectacolelor. */
```

```
public static void ordonare(int[] s, int[] t)
{
  //ordonare prin "bubble sort"

  //daca interschimbam valorile din sir
  //atunci k este 1, altfel k este 0
  int k;
  int aux;

  do
  {
    //la inceput nu sunt schimbari in siruri
    k = 0;

    for (int i = 0; i < t.length - 1; i++)
    {
      if (t[i] > t[i + 1])
      {
        //interschimbam valorile din s
        aux = s[i];
        s[i] = s[i + 1];
        s[i + 1] = aux;

        //interschimbam valorile din t
        aux = t[i];
        t[i] = t[i + 1];
        t[i + 1] = aux;

        //au fost facute schimbari in siruri
        k = 1;
      }
    }
  }
  while (k == 1);
}

/** Programul principal. */
public static void main(String[] args)
{
  //citirea numarului de spectacole
  System.out.print("Introduceti numarul de spectacole: ");
  int n = Reader.readInt();

  //citirea timpilor de incepere si terminare ai spectacolelor
  System.out.println("Introduceti timpii de incepere si " +
        "terminare ai spectacolelor:");
  int[] s = new int[n];
  int[] t = new int[n];

  for(int i = 0; i < s.length; i++)
```

```
    {
      System.out.print("s[" + i + "] = ");
      s[i] = Reader.readInt();
      System.out.print("t[" + i + "] = ");
      t[i] = Reader.readInt();
    }

    //ordonarea crescatoare a spectacolelor in functie de
    //timpul de terminare a spectacolelor
    ordonare(s, t);

    //se selecteaza prin metoda greedy spectacolele
    Vector sol = selectSpectacole(s, t);

    //afisarea rezultatelor
    System.out.println("Organizarea spectacolelor:");
    for (int i = 0; i < sol.size(); i++)
    {
      int id = ((Integer) sol.elementAt(i)).intValue();

      System.out.println(s[id] + " " + t[id]);
    }

  }
}
```

13.2 Elemente ale strategiei Greedy

Un algoritm Greedy determină o soluţie optimă a unei probleme în urma unei succesiuni de alegeri. La fiecare moment de decizie din algoritm este aleasă opţiunea care pare a fi cea mai potrivită. Această *strategie euristică* nu produce întotdeauna soluţia optimă, dar există şi cazuri când aceasta este obţinută, cum ar fi în cazul problemei selectării activităţilor. În acest paragraf vom prezenta câteva proprietăţi generale ale metodei Greedy.

Cum se poate decide dacă un algoritm Greedy poate rezolva o problemă particulară de optimizare? În general nu există o modalitate de a stabili acest lucru, dar există anumite caracteristici pe care le au majoritatea problemelor care se rezolvă prin tehnici *Greedy: proprietatea de alegere Greedy şi substructura optimă.*

În cazul general o problemă de tip Greedy, are următoarele componente:

- o *mulţime de candidaţi* (lucrări de planificat, vârfuri ale grafului etc);
- o funcţie care verifică dacă o anumită mulţime de candidaţi constituie *o soluţie posibilă* (*nu neapărat optimă*) a problemei;

- o funcţie care verifică dacă o mulţime de candidaţi este *fezabilă*, adică dacă este posibil să completăm această mulţime astfel încât să obţinem o soluţie posibilă (nu neapărat optimă) a problemei (verifică dacă planificarea este formată din activităţi care nu se suprapun, etc.);
- *o funcţie de selecţie* care indică la orice moment care este cel mai promiţător dintre candidaţii încă nefolosiţi (se alege spectacolul compatibil care se termină cel mai repede);
- *o funcţie obiectiv* care dă valoarea unei soluţii (numărul de lucrări planificate, timpul necesar executării tuturor lucrărilor într-o anumită ordine, lungimea drumului pe care l-am găsit, etc.); aceasta este funcţia pe care urmărim să o optimizăm (minimizăm/maximizăm).

Pentru a rezolva o problemă de optimizare cu Greedy, căutăm o soluţie posibilă care să optimizeze valoarea funcţiei obiectiv. Un algoritm Greedy construieşte soluţia pas cu pas. Iniţial, mulţimea candidaţilor selectaţi este vidă. La fiecare pas, încercăm să adăugăm acestei mulţimi cel mai promiţător candidat, conform funcţiei de selecţie. Dacă, după o astfel de adăugare, mulţimea de candidaţi selectaţi nu mai este fezabilă, eliminăm ultimul candidat adăugat, iar acesta nu va mai fi niciodată considerat. Dacă, după adăugare, mulţimea de candidaţi selectaţi este fezabilă, ultimul candidat adăugat va rămâne de acum încolo în ea. De fiecare dată când lărgim mulţimea candidaţilor selectaţi, verificăm dacă această mulţime nu constituie o soluţie posibilă a problemei noastre. Dacă algoritmul Greedy funcţionează corect, prima soluţie găsită va fi totodată o soluţie optimă a problemei. Soluţia optimă nu este în mod necesar unică: se poate ca funcţia obiectiv să aibă aceeaşi valoare optimă pentru mai multe soluţii posibile. Descrierea în pseudocod a unui algoritm Greedy general este:

```
funcţie greedy(C) // C este mulţimea candidaţilor
 S ←
ϕ // S este mulţimea în care construim soluţia
 cât timp not soluţie(S) şi C ≠ ϕ
   x← un element din C care maximizează select(x)
   C ← C − {x}
   dacă fezabil(S ∪ {x}) atunci S ← S ∪ {x}
 dacă soluţie(S) atunci
   return S
 altfel
   return "nu există soluţie"
```

Este de înţeles acum de ce un astfel de algoritm se numeşte "lacom" (am putea să-l numim şi "nechibzuit"). La fiecare pas, *greedy()* alege cel mai bun candidat la momentul respectiv, fără să-i pese de viitor şi fără să se răzgândească. Dacă un candidat este inclus în soluţie, el rămâne acolo; dacă un candidat este exclus din soluţie, el nu va mai fi niciodată reconsiderat. Asemenea unui întreprinzător care urmăreşte câştigul imediat în dauna celui de perspectivă, un algoritm Greedy acţionează simplist. Totuşi, ca şi în afaceri, o astfel de metodă poate da rezultate foarte bune tocmai datorită simplităţii ei. Funcţia de selectare este în general derivată, ca şi funcţia de continuare de la metoda backtracking, din funcţia obiectiv. Să identificăm acum elementele strategiei Greedy pentru următoarea problemă:

Să se scrie un algoritm care este capabil să dea o anumită sumă, reprezentând restul unui client, folosind un număr cât mai mic de monezi, având valorile de 1,5 şi 25 de unităţi.

Elementele strategiei Greedy sunt:

- *Candidaţii*: mulţimea iniţială de monezi de 1, 5 şi 25 unităţi, în care presupunem că din fiecare tip de monedă avem o cantitate nelimitată;
- *O soluţie posibilă*: valoarea totală a unei astfel de mulţimi de monezi selectate trebuie să fie exact valoarea pe care trebuie să o dăm ca rest;
- *O mulţime fezabilă*: valoarea totală a unei astfel de mulţimi de monezi selectate nu este mai mare decât valoarea pe care trebuie să o dăm ca rest;
- *Funcţia de selecţie*: se alege cea mai mare monedă din mulţimea de candidaţi rămasă;
- *Funcţia obiectiv*: numărul de monezi folosite în soluţie; se doreşte minimizarea acestui număr.

Se poate demonstra că algoritmul Greedy va găsi în acest caz mereu soluţia optimă - restul cu un număr minim de monezi. Pe de altă parte, presupunând că există şi monezi de 12 unităţi sau că unele din tipurile de monezi lipsesc din mulţimea iniţială de candidaţi, se pot găsi contraexemple pentru care algoritmul nu găseşte soluţia optimă, sau nu găseşte nici o soluţie cu toate că există una.

Evident, soluţia optimă se poate găsi şi încercând toate combinările posibile de monezi (folosind *backtracking*), dar soluţia ar avea în acest caz o complexitate exponenţială, în timp ce complexitatea algoritmului Greedy este liniară.

Un algoritm Greedy nu duce deci întotdeauna la soluţia optimă sau la o soluţie. Este doar un principiu general, urmând ca pentru fiecare caz în parte să determinăm dacă obţinem sau nu soluţia optimă.

Să vedem acum care sunt cele două caracteristici esenţiale pe care trebuie să le respecte o problemă pentru a putea fi abordată folosind Greedy.

13.2.1 Proprietatea de alegere Greedy

Prima caracteristică a unei probleme de tip Greedy este *proprietatea alegerii Greedy*, adică se poate ajunge la o soluţie optimă global, realizând alegeri (Greedy) *optime local*. În procesul de construcţie a unei soluţii din cadrul unui algoritm Greedy se realizează o alegere care pare a fi cea mai bună la momentul respectiv. Alegerea realizată de un algoritm Greedy poate depinde de alegerile făcute până în acel moment, dar nu ia în calcul niciodată alegerile care pot fi făcute ulterior.

Desigur, trebuie să demonstrăm că o alegere Greedy la fiecare pas conduce la o soluţie optimă global, dar aceasta este o problemă mai delicată. De obicei, demonstraţia examinează o soluţie optimă global. Apoi se arată că soluţia poate fi modificată astfel încât la fiecare pas este realizată o alegere Greedy, iar această alegere reduce problema la una similară dar de dimensiuni mai reduse. Se aplică apoi principiul inducţiei matematice pentru a arăta că o alegere Greedy poate fi utilizată la fiecare pas. Faptul că o alegere Greedy conduce la o problemă de dimensiuni mai mici reduce demonstraţia corectitudinii la demonstrarea faptului că o soluţie optimă trebuie să evidenţieze o substructură optimă.

13.2.2 Substructură optimă

O problemă evidenţiază o s*ubstructură optimă* dacă o soluţie optimă a problemei conţine soluţii optime ale subproblemelor. Această proprietate este cheia pentru aplicarea programării dinamice sau a unui algoritm Greedy. Ca exemplu al unei structuri optime, să ne reamintim demonstraţia corectitudinii algoritmului pentru problema selectării spectacolelor, unde se arată că dacă o soluţie optimă A a problemei selectării activităţilor începe cu activitatea 1, atunci mulţimea activităţilor $A' = A - \{1\}$ este o soluţie optimă pentru problema selectării activităţilor $S' = \{i \in S | s_i \geq t_1\}$.

Cu proprietatea de substructură optimă ne vom întâlni în capitolul următor, în care vom prezenta cum se pot rezolva problemele care o respectă (fără a respecta şi proprietatea de alegere Greedy) folosind metoda programării dinamice.

13.3 Minimizarea timpului mediu de aşteptare

O singură staţie de servire (procesor, pompă de benzină etc) trebuie să satisfacă cererile a n clienţi. Timpul de servire necesar fiecărui client este cunoscut în prealabil: pentru clientul i este necesar un timp $t_i, i = 1, n$. *Dorim să minimizăm timpul total de aşteptare:*

$$T = \sum_{i=1}^{n} (\text{timpul de aşteptare pentru clientul } i)$$

Ceea ce este acelaşi lucru cu a minimiza timpul mediu de aşteptare, care este $\frac{T}{n}$.

De exemplu, dacă avem trei clienţi cu $t_1 = 5, t_2 = 10, t_3 = 3$, sunt posibile şase ordini de servire:

Ordine			Timpul (T)
1	2	3	5+(5+10)+(5+10+3)
1	3	2	5+(5+3)+(5+3+10)
2	1	3	10+(10+5)+(10+5+3)
2	3	1	10+(10+3)+(10+3+5)
3	1	2	3+(3+5)+(3+5+10) optim
3	2	1	3+(3+10)+(3+10+5)

În primul caz, clientul 1 este servit primul, clientul 2 aşteaptă până este servit clientul 1 şi apoi este servit, clientul 3 aşteaptă până sunt serviţi clienţii 1, 2 şi apoi este servit. Timpul total de aşteptare a celor trei clienţi este 38.

Timpul total minim de aşteptare este obţinut în al cincilea caz şi are valoarea 29.

Algoritmul Greedy este foarte simplu - la fiecare pas se selectează clientul cu timpul minim de servire din mulţimea de clienţi rămasă. Vom demonstra că acest algoritm este optim. Fie $I = (i_1\, i_2\, \ldots\, i_n)$ o permutare oarecare a întregilor $\{1, 2, \ldots, n\}$. Dacă servirea are loc în ordinea *I*, avem:

$$T(I) = t_{i_1} + (t_{i_1} + t_{i_2}) + (t_{i_1} + t_{i_2} + t_{i_3}) + \ldots = nt_{i_1} + (n-1)t_{i_2} + \ldots = \sum_{k=1}^{n} (n-k+1)t_{i_k}$$

Presupunem acum că *I* este astfel ales încât putem găsi doi întregi $a < b$ cu

$$t_{i_a} > t_{i_b}$$

deci există un client (b) care necesită un timp mai lung de deservire, şi care este servit înainte (de a).

Interschimbăm pe i_a cu i_b în *I*; cu alte cuvinte, clientul care a fost servit al b-lea va fi servit acum al a-lea şi invers. Obţinem o nouă ordine de servire *I'*, care este de preferat deoarece

$$T(I') = (n-a+1)t_{i_b} + (n-b+1)t_{i_a} + \sum_{k=1,k\neq a,b}^{n} (n-k+1)t_{i_k}$$

$$T(I) - T(I') = (n-a+1)(t_{i_a} - t_{i_b}) + (n-b+1)(t_{i_b} - t_{i_a}) = (b-a)(t_{i_a} - t_{i_b}) > 0$$

Aplicând succesiv pasul de mai sus se obţine o permutare optimă, de tipul $J = (j_1, j_2, \ldots, j_n)$ pentru care avem:

$$t_{j_1} \leq t_{j_2} \leq \cdots \leq t_{j_n}.$$

Prin metoda Greedy, selectând permanent clientul cu timpul cel mai mic de deservire, obţinem deci întotdeauna planificarea optimă a clienţilor. Problema poate fi generalizată şi pentru un sistem cu mai multe staţii de servire.

Implementarea algoritmului se reduce la o banală ordonare a clienţilor crescător după timpul de deservire şi este prezentată în **Listing 13.2.**

Listing 13.2: Soluţia problemei minimizării timpului de aşteptare

```
import java.io.*;
import io.Reader;

/**
 * Program pentru minimizarea timpului de asteptare al unui
 * client pentru a fi deservit de o statie (metoda Greedy).
 */
public class MinimTimp
{
  /** Ordonarea timpilor de asteptare ai clientilor. */
  public static void ordonare(int[] t)
  {
    //ordonare prin "bubble sort"

    //daca interschimbam valorile din sir
    //atunci k este 1, altfel k este 0
    int k;
    int aux;

    do
    {
      //la inceput nu sunt schimbari in siruri
      k = 0;

      for (int i = 0; i < t.length - 1; i++)
      {
        if (t[i] > t[i + 1])
        {
          //interschimbam valorile din t
          aux = t[i];
```

```
          t[i] = t[i + 1];
          t[i + 1] = aux;

          //au fost facute schimbari in sir
          k = 1;
        }
      }
    }
    while (k == 1);
  }

  /** Calculeaza timpul total minim de asteptare.*/
  public static int calculTimpMinim(int[] t)
  {
    int s = 0;

    System.out.println("Ordinea de deservire este:");

    for (int i = 1; i <= t.length; i++)
    {
      s += (t.length - i + 1) * t[i - 1];

      System.out.print(t[i - 1] + " ");
    }
    System.out.println();

    return s;
  }


  /** Programul principal.*/
  public static void main(String[] args)
  {
    //citirea timpului de asteptare al fiecarui client
    System.out.println("Timpii de asteptare pentru " +
      "clienti (pe aceeasi linie):");
    int[] t = Reader.readIntArray();

    //ordonarea timpilor de asteptare ai clientilor
    ordonare(t);

    System.out.println(
      "Timpul total de asteptare are valoarea minima = " +
      calculTimpMinim(t));

  }
}
```

13.4 Interclasarea optimă a mai multor şiruri ordonate

Să presupunem că avem două şiruri S_1 şi S_2, de lungime m şi n, ordonate crescător şi că dorim să obţinem prin interclasarea lor un şir ordonat crescător, S, care conţine exact elementele din cele două şiruri. Dacă interclasarea are loc prin plasarea elementelor din S_1 şi S_2 în noul şir, S, atunci numărul deplasărilor este $m + n$.

Generalizând, să considerăm acum n şiruri $S_1, S_2, \ldots S_n$, fiecare şir $S_i, i = 1, n$, fiind format din q_i elemente ordonate crescător (vom numi q_i *lungimea* lui S_i). Ne propunem să obţinem şirul S ordonat crescător, conţinând exact elementele din cele n şiruri. Vom realiza acest lucru prin interclasări succesive de câte două şiruri. Problema interclasării optime a mai multor şiruri ordonate constă în determinarea *ordinii* optime în care trebuie efectuate aceste interclasări, astfel încât numărul total de operaţii (deplasări) să fie cât mai mic. Exemplul de mai jos ne arată că problema astfel formulată nu este banală, adică nu este indiferent în ce ordine se fac interclasările.

Exemplu: Fie şirurile S_1, S_2, S_3 de lungimi $q_1 = 30, q_2 = 20, q_3 = 10$. Dacă interclasăm pe S_1 cu S_2, iar rezultatul îl interclasăm cu S_3, numărul total al deplasărilor este $(30 + 20) + (50 + 10) = 110$. Dacă interclasăm pe S_3 cu S_2, iar rezultatul îl interclasăm cu S_1, numărul total al deplasărilor este $(10 + 20) + (30 + 30) = 90$, deci cu 20 de operaţii mai puţin.

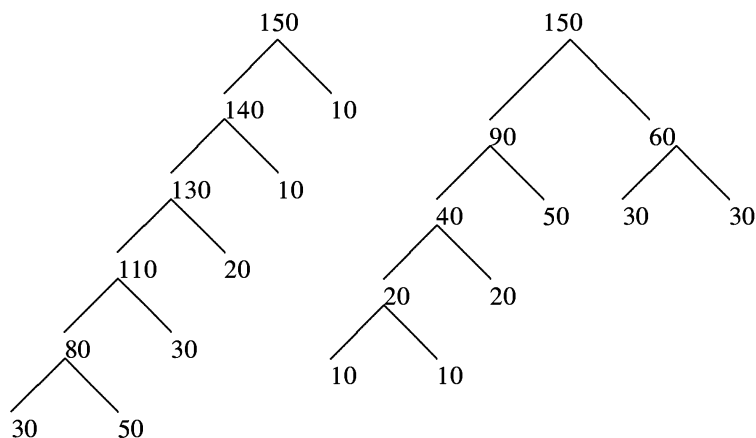
Ataşăm fiecărei strategii de interclasare câte un arbore binar în care valoarea fiecărui vârf este dată de lungimea şirului pe care îl reprezintă. Dacă şirurile $S_1, S_2, \ldots, S_6$ au lungimile $q_1 = 30, q_2 = 10, q_3 = 20, q_4 = 30, q_5 = 50, q_6 = 10$, două astfel de strategii de interclasare sunt reprezentate prin arborii din **Figura 13.1**.

Observăm că fiecare arbore are 6 vârfuri terminale, corespunzând celor 6 şiruri iniţiale şi 5 vârfuri neterminale, corespunzând celor 5 interclasări care definesc strategia respectivă. Numerotăm vârfurile în felul următor: vârful terminal i ($i \in 1, 2, 3, 4, 5, 6$), va corespunde şirului S_i, iar vârfurile neterminale se numerotează de la 7 la 11 în ordinea obţinerii interclasărilor respective (**Figura 13.2**).

Strategia Greedy apare în **Figura 13.1** (arborele din partea dreaptă) şi constă în a interclasa mereu cele mai scurte două şiruri disponibile la momentul respectiv.

Pentru a interclasa şirurile $S_1, S_2, \ldots, S_n$, de lungimi $q_1, q_2, \ldots, q_n$, obţinem pentru fiecare strategie câte un arbore binar cu n vârfuri terminale numerotate

Figura 13.1: Reprezentarea strategiilor de interclasare.



de la 1 la n şi $n-1$ vârfuri neterminale numerotate de la $n+1$ la $2n-1$. Definim pentru un arbore oarecare *A* de acest tip lungimea externă ponderată:

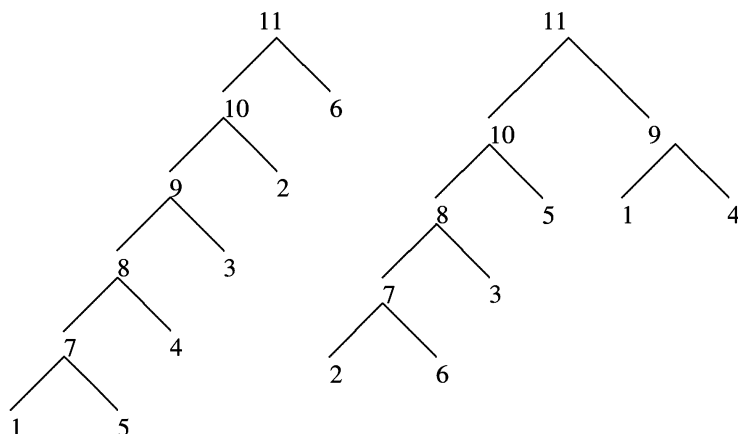
$$L(A) = \sum_{i=1}^{n} a_i q_i$$

unde a_i este adâncimea vârfului *i*. Este uşor de observat că numărul total de deplasări de elemente pentru strategia corespunzătoare lui *A* este chiar $L(A)$. Soluţia optimă a problemei noastre este atunci arborele (strategia) pentru care lungimea externă ponderată este minimă.

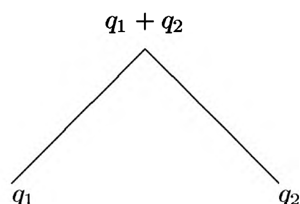
Teorema 13.4.1 *Prin metoda Greedy, în care se interclasează la fiecare pas cele două şiruri de lungime minimă, se obţine şirul s cu un număr minim de operaţii.*

Demonstraţie: Demonstrăm prin inducţie. Pentru $n = 1$, proprietatea este verificată. Presupunem că proprietatea este adevărată pentru $n-1$ şiruri. Fie *A*

Figura 13.2: Numerotarea vârfurilor arborilor din **Figura** 13.1



arborele strategiei Greedy de interclasare a n şiruri de lungime $q_1 \leq q_2 \leq \ldots \leq q_n$. Fie B un arbore cu lungimea externă ponderată minimă, corespunzător unei strategii optime de interclasare a celor n şiruri. În arborele A apare subarborele:



reprezentând prima interclasare făcută conform strategiei Greedy. În arborele B, fie un vârf neterminal de adâncime maximă. Cei doi fii ai acestui vârf sunt atunci două vârfuri terminale q_j şi q_k. Fie B' arborele obţinut din B schimbând între ele vârfurile q_1 şi q_j, respectiv q_2 şi q_k. Evident, $L(B') \leq L(B)$. Deoarece B are lungimea externă ponderată minimă, rezultă că $L(B) = L(B')$. Eliminând din B' vârfurile q_1 şi q_2, obţinem un arbore B" cu $n - 1$ vârfuri

terminale $q_1 + q_2, q_3, \ldots, q_n$. Arborele B' are lungimea externă ponderată minimă şi $L(B') = L(B) + (q_1 + q_2)$. Rezultă că şi B are lungimea externă ponderată minimă. Atunci, conform ipotezei inducţiei, avem $L(B) = L(A')$ unde A' este arborele strategiei Greedy de interclasare a şirurilor de lungime $q_1 + q_2, q_3, \ldots, q_n$. Cum A se obţine din A' ataşând la vârful $q_1 + q_2$ fiii q_1 şi q_2, iar B' se obţine în acelaşi mod din B'', rezultă că $L(A) = L(B') = L(B)$. Proprietatea este deci adevărată pentru orice n.

Deoarece algoritmul alege la fiecare pas şirurile disponibile de lungime minimă, structura de date adecvată pentru a reţine şirurile este coada de priorităţi, prezentată în paragraful 10.7. Implementarea algoritmului este simplă şi o lăsăm ca exerciţiu.

Rezumat

În acest capitol am prezentat metoda Greedy, care se poate aplica problemelor care respectă substructura optimă şi principiul alegerii Greedy. În cazul acestei metode, soluţia se construieşte succesiv, la fiecare pas realizând o alegere optimă local, fără a reveni asupra deciziilor anterioare. Soluţia construită astfel, va fi o soluţie optimă a problemei de rezolvat. Am văzut modul în care se poate demonstra corectitudinea unui algoritm utilizând un procedeu general, care se poate aplica la mulţi algoritmi de acest tip.

Noţiuni fundamentale

funcţie de selecţie: funcţie care indică cel mai promiţător dintre candidaţii încă nefolosiţi la un moment dat.

proprietatea de alegere Greedy: se poate ajunge la o soluţie optimă global, realizând alegeri (Greedy) *optime local.*

problemă de optimizare: problemă în care se cere minimizarea sau maximizarea unei funcţii obiectiv.

substructură optimă (principiul optimalităţii): o problemă evidenţiază o *substructură optimă* dacă o soluţie optimă a problemei conţine soluţii optime ale subproblemelor.

Erori frecvente

1. Nu încercaţi să aplicaţi această metodă orbeşte, fără a verifica dacă problema respectă substructura optimă şi proprietatea de alegere Greedy.

2. Faptul că aţi găsit un algoritm Greedy care funcţionează corect pe unele exemple, nu înseamnă că metoda va da soluţia optimă pentru orice date de intrare. Pentru a fi siguri de aceasta, demonstraţi corectitudinea algoritmului.

Exerciţii

Teorie

1. Calculaţi complexitatea rezolvării problemei spectacolelor din paragraful 13.1, **Listing 13.1**.

2. Găsiţi o modalitate de a reduce complexitatea rezolvării problemei anterioare la $O(n \log n)$.

3. În rezolvarea problemei interclasării optime din paragraful 13.4 am recomandat utilizarea unei cozi de priorităţi. Care este complexitatea algoritmului în acest caz? Dar dacă în loc de o coadă de priorităţi folosim un arbore binar de căutare?

În practică

1. Rezolvarea problemei spectacolelor din paragraful 13.1 foloseşte metoda bulelor pentru a ordona spectacolele în ordinea crescătoare a timpului de terminare. Înlocuiţi metoda bulelor cu o ordonare mai eficientă.

2. Rezolvaţi problema interclasării optimale din paragraful 13.4 folosind

   (a) O listă înlănţuită;

   (b) O coadă de priorităţi;

   (c) Un arbore binar de căutare.

3. (*Problema rucsacului*) Avem la dispoziţie un rucsac de capacitate M şi n obiecte diferite (câte unul din fiecare) cu costurile c_i şi greutatea g_i. Scrieţi un algoritm care aşează aceste obiecte în rucsac astfel încât costul total să fie maxim. Suma greutăţilor obiectelor din rucsac nu poate depăşi capacitatea rucsacului. Dacă un obiect nu încape în rucsac, se poate lua doar o parte (fracţiune) din el.

   Indicaţie: Este uşor de intuit că pentru obţinerea unui profit (cost) total maxim trebuie alese obiecte de greutate mică şi cost mare. Demonstraţia riguroasă a afirmaţiei anterioare v-o propunem spre rezolvare.

4. (*Problema discretă a rucsacului*) Acelaşi enunţ ca la problema precedentă, cu diferenţa că dintr-un obiect nu se poate pune o fracţiune (un obiect fie se pune întreg, fie nu se pune). Arătaţi că algoritmul Greedy de la problema precedentă nu furnizează întotdeauna soluţie optimă în acest caz. Găsiţi un algoritm care furnizează întotdeauna soluţie optimă! Ce complexitate are acest algoritm?

   Indicaţie: Soluţia optimă pentru această problemă poate fi determinată doar prin backtracking sau programare dinamică (prezentată în capitolul următor). De exemplu, pentru c=(5,3,3), g=(3,2,2) şi M=4, prin aplicarea algoritmului de la problema precedentă am selecta doar obiectul 1 (obiectul 2 nu ar încape întreg în rucsac deci nu ar putea fi selectat) şi am obţine costul 5, în timp ce dacă am selecta obiectele 2 şi 3 am obţine costul 6.

5. Găsiţi o soluţie Greedy pentru problema comis-voiajorului propusă la capitolul 11. Este această soluţie optimă?

   Indicaţie: Conform strategiei greedy, vom construi ciclul pas cu pas, adăugând la fiecare iteraţie cea mai scurtă muchie disponibilă cu următoarele proprietăţi:

   - *nu formează un ciclu cu muchiile deja selectate (exceptând pentru ultima muchie aleasă, care completează ciclul);*
   - *nu există încă două muchii deja selectate, astfel încât cele trei muchii să fie incidente în acelaşi vârf.*

   Acest algoritm nu numai că nu furnizează soluţia optimă, dar în multe situaţii nici măcar nu găseşte o soluţie.

14. Programare dinamică

> Mulţi oameni preferă să tolereze o problemă pe care nu o pot rezolva, decât o soluţie pe care nu o pot înţelege.
>
> Woolsey and Swanson

În capitolul anterior am văzut cum putem aplica metoda Greedy problemelor de optimizare care respectă principiul optimalităţii (substructură optimă) şi principiul alegerii Greedy. Programarea dinamică este o altă metodă de elaborare a algoritmilor care se aplică problemelor de optimizare care respectă principiul optimalităţii (fără a respecta principiul alegerii Greedy). În acest capitol vom introduce gradat noţiunile şi tehnicile specifice acestei metode de elaborare a algoritmilor, considerată de mulţi ca având un grad de dificultate mai mare decât celelalte metode[1]. Dar nu vă îngrijioraţi! Materialul din acest capitol poate fi cu uşurinţă parcurs de către oricine, datorită structurării gradate a noţiunilor prezentate.

Pe parcursul acestui capitol vom vedea:

- Cum se tratează *dinamic* problemele de recurenţă matematică, în vederea unei rezolvări eficiente;
- Ce este principiul optimalităţii;
- Cum se aplică principiul optimalităţii pentru a descompune problema de programare dinamică în subprobleme;
- Cum se creează relaţiile de recurenţă care rezolvă problema de programare dinamică;

[1] De altfel, majoritatea problemelor spinoase alese pentru concursurile naţionale şi internaţionale de informatică se rezolvă prin această metodă.

- Cum se poate reface soluţia problemei pe baza calculelor efectuate pentru obţinerea valorii optime;
- Câteva probleme clasice de programare dinamică.

14.1 Istoric şi descriere

Programarea dinamică provine din domeniul cercetărilor operaţionale, unde constituie un domeniu de sine stătător. Cuvântul *programare* din numele acestei metode nu are nimic de a face cu scrierea de programe pentru calculator. Matematicienii foloseau acest cuvânt pentru a descrie un set de reguli pe care oricine le poate urmări pentru a rezolva un anumit tip de problemă. Programarea dinamică a fost introdusă în anii 1950 de către matematicianul Richard Bellman, care a descris modalitatea prin care se rezolvă problemele în care trebuie luată o decizie optimală la fiecare pas. În cei mai bine de 50 de ani de la apariţia acestei metode utilizarea şi aplicarea programării dinamice au crescut enorm.

Programarea dinamică descompune de obicei problema de optimizare originală în subprobleme şi alege cele mai bune soluţii pentru subprobleme începând chiar cu cele de dimensiune minimă. Soluţia optimă a problemelor de dimensiune mai mare este obţinută pe baza soluţiilor optime a problemelor de dimensiune mai mică cu ajutorul unei formule de recurenţă care face legătura între soluţii. Până în acest punct descrierea făcută nu diferă cu nimic de cea a tehnicii Divide et Impera. Ceea ce face ca programarea dinamică să fie mai specială este faptul că formula de recurenţă este folosită pentru a elimina toate soluţiile subproblemelor care nu pot genera soluţia optimă. Mai mult, în cazul programării dinamice se reţin soluţiile subproblemelor care pot fi utilizate în calculul soluţiei optime.

Similarităţi există şi cu metoda Greedy. Aşa cum vom vedea, şi problemele de programare dinamică trebuie să respecte *substructura optimă*, numită în această situaţie *principiul optimalităţii*. Diferenţa constă în faptul că în cazul programării dinamice nu se face o alegere Greedy, mergând pe optimul local, ci se iau în considerare *toate* soluţiile optimale ale subproblemelor cu un ordin de mărime mai mic. Există multe probleme care admit o rezolvare atât prin metoda Greedy, cât şi prin programare dinamică (problema rucsacului, problema comis-voiajorului). În aceste situaţii, soluţia Greedy va avea o complexitate în timp mai mică, dar soluţiile găsite nu vor fi întotdeauna cele optime, în timp ce soluţiile date prin programare dinamică for avea o complexitate în timp ceva mai mare, dar vor furniza soluţia optimă.

Dificultăţile care apar în utilizarea metodei programării dinamice au făcut ca această metodă să fie considerată de mulţi ca fiind prea complexă şi în con-

secinţă să evite aprofundarea şi utilizarea ei. Scopul acestui capitol este tocmai acela de a forma treptat capacitatea de a analiza şi rezolva cu multă uşurinţă orice gen de problemă de programare dinamică.

14.2 Primii paşi în programarea dinamică

Acest paragraf este consacrat tratării unor probleme de recurenţă matematică. Nu putem afirma că problemele de recurenţă matematică se încadrează strict în categoria problemelor de programare dinamică, deoarece ele nu sunt probleme de optimizare. Totuşi, modul în care vom rezolva aici recurenţele introduce deja un concept esenţial pentru rezolvarea problemelor de programare dinamică: tabelul[2]. Tot în acest paragraf vom analiza şi care sunt avantajele utilizării tabelelor pentru a reţine soluţiile subproblemelor în dauna soluţiilor recursive.

14.2.1 Probleme de recurenţă matematică tratate dinamic

Şirul lui Fibonacci

Un exemplu clasic, adesea întâlnit în lucrările care tratează recursivitatea, prezentat şi în paragraful 12.1.1 (pagina 138) este calculul şirului lui Fibonacci, definit astfel:

$F_n = F_{n-1} + F_{n-2};\ F_0 = 0;\ F_1 = 1$

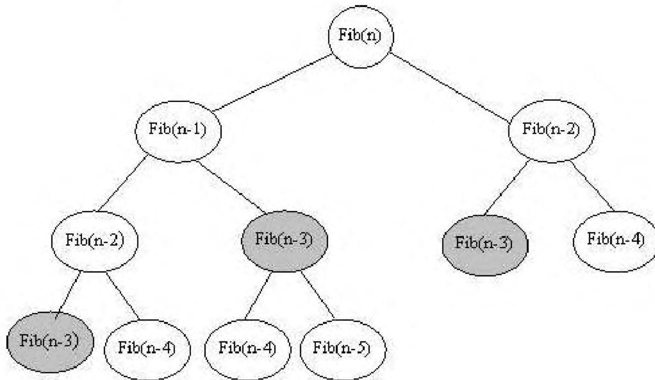
Metoda recursivă care calculează F_n este descrisă în **Listing 14.1**. Aşa cum am văzut deja în paragraful 9.4.3 (pagina 26), funcţia de mai sus recalculează din nou şi din nou termenii şirului, rezultând astfel un timp de lucru exponenţial ($O(\phi^n)$). **Figura** 14.1 pune în evidenţă modul în care se recalculează din nou şi din nou aceleaşi valori din cauza apelurilor recursive suprapuse.

Listing 14.1: Metoda recursivă de calcul a şirului lui Fibonacci

```
public static final int fibonacciRec(int n)
{
  if (n == 0 || n == 1)
  {
    return n;
  }
  else
  {
```

[2]Este interesant de remarcat faptul că recurgerea la tabele a dat şi numele metodei. Astfel, în cercetările operaţionale (domeniul din care provine această metodă) termenul "programare" se referă la un set de reguli care se aplică pe un tabel şi nu la scrierea unui program pentru calculator.

Figura 14.1: Reprezentarea arborescentă a apelurilor recursive realizate pentru a calcula şirul lui Fibonacci. Se observă că `Fib(n-3)` este calculat de 3 ori.




```
    return fibonacciRec(n - 1) + fibonacciRec(n - 2);
  }
}
```

Putem evita cu uşurinţă recalcularea termenilor şirului, calculând elementele şirului succesiv, de la F_1 la F_n şi introducându-le într-un tabel (cu o singură linie) pe măsură ce sunt calculate (**Listing 14.2**). Această idee, banală în aparenţă, reduce complexitatea algoritmului de la $O(\phi^n)$ la $O(n)$ şi introduce un concept fundamental pentru rezolvarea problemelor de programare dinamică: tabelul în care se reţin soluţiile subproblemelor pentru a evita recalcularea lor.

Listing 14.2: Metoda iterativă de calcul a şirului lui Fibonacci

```
public static final int fibonacciIter(int n)
{
  int[] fib = (n >= 2) ? new int[n+1] : new int[2];
  fib[0] = 0;
  fib[1] = 1;

  for (int i = 2; i <= n; ++i)
  {
    fib[i] = fib[i-1] + fib[i-2];
  }
  return fib[n];
}
```

Listing 14.3: Exemplu simplu de utilizare a metodei `fibonacciIter` pentru a calcula termenul de ordin `n` din cadrul şirului lui Fibonacci

```
import java.io.*;
import io.Reader;

public class Fibonacci
{
  public static int fibonacciIter(int n)
  {
    //...
  }

  public static void main(String args[])
  {
    System.out.print("n = ");
    int n = Reader.readInt();

    if (n < 1) return;

    System.out.println("Termenul " + n + " din sirul" +
      " Fibonacci este: " + fibonacciIter(n - 1));
  }
}
```

Clasa `Fibonacci` din **Listing 14.3** citeşte un număr întreg `n` de la tastatură şi calculează valoarea şirului lui Fibonacci la poziţia `n` (altfel spus, termenul de ordin `n` al şirului) folosind metoda `fibonacciIter()`.

Calculul combinărilor

În cadrul calculului şirului lui Fibonacci din paragraful anterior am utilizat un tabel unidimensional pentru a reţine valorile şirului. În exemplul care urmează vom prezenta o problemă în rezolvarea căreia este necesară utilizarea unui tablou bidimensional (utilizarea tablourilor bidimensionale este mai frecventă în rezolvarea problemelor de programare dinamică).

Este cunoscut faptul că prin combinări de n luate câte k (notate $C(n, k)$) se înţelege numărul de submulţimi care conţin k elemente ale unei mulţimi cu n elemente. Formula matematică pentru combinări de n luate câte k este:

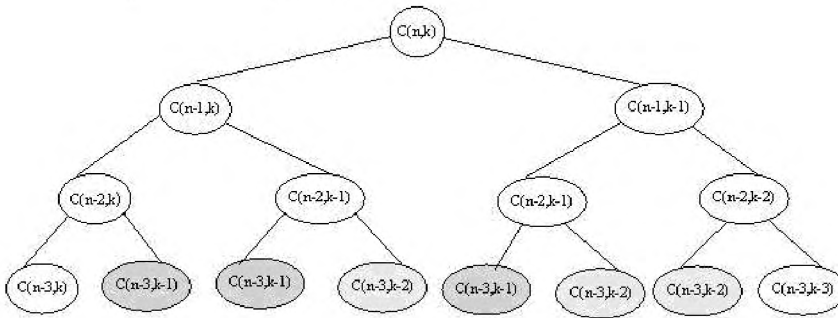
$$C(n, k) = \frac{n!}{k!(n-k)!}$$

Nu este însă avantajos ca numerele $C(n, k)$ să fie calculate direct, folosind formula de mai sus. Formula de recurenţă $C(n, k) = C(n-1, k) + C(n-1, k-1)$ este mult mai rapidă, deoarece utilizează numai adunări şi elimină operaţiile de împărţire şi înmulţire care sunt mai costisitoare în timp. Metoda

Listing 14.4: Calculul recursiv al combinărilor

```
public static int combRec(int n, int k)
{
  if (n == k || k == 0)
  {
    return 1;
  }
  else
  {
    return combRec(n - 1, k) + combRec(n - 1, k - 1);
  }
}
```

Figura 14.2: Reprezentarea arborelui rezultat în urma apelurilor recursive generate de recurenţa pentru calculul combinărilor (se observă recalcularea inutilă a valorilor $C(n-3, k-1)$ şi $C(n-3, k-2)$).



`combRec()` din **Listing 14.4** reprezintă implementarea recursivă directă a formulei de recurenţă anterioare.

Această metodă, deşi este foarte elegantă şi compactă, suferă de aceleaşi recalculări inutile ale anumitor valori ca şi algoritmul recursiv pentru calculul şirului lui Fibonacci din **Listing 14.1**. Recalculările inutile sunt puse în evidenţă de **Figura 14.2**, în care am reprezentat primele 3 nivele din arborele generat prin calculul recursiv al combinărilor.

Notând cu t_{nk} numărul de operaţii efectuate pentru a calcula $C(n, k)$ folosind metoda de mai sus, obţinem relaţia de recurenţă:

$$t_{nk} = t_{n-1k} + t_{n-1k-1}; \quad t_{n0} = t_{nn} = 1.$$

Iterând recurenţa de mai sus se obţine cu uşurinţă faptul că timpul de calcul

Figura 14.3: Tabelul utilizat pentru calculul iterativ al combinărilor. Se observă că acest tabel nu este altceva decât triunghiul lui Pascal.

$C(0,0)$				
$C(1,0)$	$C(1,1)$			
$C(2,0)$	$C(2,1)$	$C(2,2)$		
$C(3,0)$	$C(3,1)$	$C(3,2)$	$C(3,3)$	
$C(4,0)$	$C(4,1)$	$C(4,2)$	$C(4,3)$	$C(4,4)$

Listing 14.5: Calculul iterativ al combinărilor

```
public static int combIter(int n, int k)
{
  int[][] comb = new int[n + 1][n + 1];
  for (int i = 0; i <= n; ++i)
  {
    comb[i][0] = comb[i][i] = 1;
    for (int j = 1; j < i; ++j)
    {
      comb[i][j] = comb[i - 1][j] + comb[i - 1][j - 1];
    }
  }
  return comb[n][k];
}
```

pentru t_{nk} este în $O(2^n)$.

Pentru a evita repetarea inutilă a calculelor, vom folosi din nou un tabel în care în celula (i, j) se va păstra valoarea lui $C(i, j)$ (indicele i trebuie să fie mai mare sau egal cu j, deci tabelul nostru va avea elemente doar sub diagonala principală). Examinând **Figura** 14.3 se constată că tabelul construit de noi nu este altceva decât triunghiul lui Pascal.

Metoda `combIter()` din **Listing 14.5** realizează calculul iterativ al combinărilor folosind un tabel (triunghiul lui Pascal). Complexitatea acestei metode este $O(n^2)$, deci ea este net superioară ca şi eficienţă metodei `combRec()` din **Listing 14.4**.

În cazul şirului lui Fibonacci am evaluat termenii începând cu cei de grad inferior şi terminând cu cei de grad superior. În cazul combinărilor am calculat treptat mai întâi combinări de un element, apoi combinări de două elemente etc., deoarece $C(n, k)$ depinde direct de $C(n-1, k)$ şi $C(n-1, k-1)$, deci de două valori aflate pe linia anterioară în tabelul reprezentând triunghiul lui Pascal. O idee importantă care reiese de aici cu uşurinţă este următoarea:

"întotdeauna în cazul unei probleme de programare dinamică este necesar ca în momentul în care se rezolvă o problemă, toate subproblemele ei să fi fost deja rezolvate."

Strategia de rezolvare pentru problemele de programare dinamică decurge astfel normal din această observaţie:

- se rezolvă mai întâi problemele de dimensiune mică pentru care soluţia este evidentă (în cazul nostru acestea sunt $F_0 = 0$, $F_1 = 1$, respectiv $C(i, 0) = 1$);

- pe baza suproblemelor de dimensiune inferioară se rezolvă subproblema cu un ordin de dimensiune mai mare (în cazul nostru, pe baza lui F_{i-1} şi F_{i-2} se calculează F_i, respectiv pe baza lui $C(i-1, j)$, $C(i-1, j-1)$ se calculează $C(i, j)$).

Clasa `CombinariIt` din **Listing 14.6** reprezintă un exemplu de simplu de utilizare a metodei `combIter()` pentru a calcula combinările a două numere citite de la tastatură.

Problema parantezelor

Cele două probleme prezentate anterior sunt relativ simple, deoarece recurenţa care trebuie rezolvată apare explicit în enunţ. Atât pentru şirul lui Fibonacci, cât şi pentru calculul combinărilor, ni se furnizează explicit relaţia de recurenţă pe care trebuie să o rezolvăm. Problema parantezelor este un exemplu de problemă în care recurenţa care trebuie rezolvată nu apare explicit în enunţ. Rămâne în sarcina noastră să construim relaţia de recurenţă care rezolvă această problemă. Din acest punct de vedere, problema parantezelor se apropie mai mult de problemele reale de programare dinamică, deoarece nici acolo recurenţa care trebuie rezolvată nu este dată explicit. Mai mult, am putea chiar afirma că principala dificultate în cazul problemelor de programare dinamică constă, mai ales la începători, în formularea corectă a relaţiei de recurenţă care rezolvă problema. Aşadar, dificultatea principală în cazul problemelor de programare dinamică este construirea recurenţei, scrierea programului pentru rezolvarea propriu-zisă a recurenţei fiind în cele mai multe cazuri banală.

Enunţul problemei parantezelor este simplu:

Dându-se un număr natural $n > 0$, se cere să se determine numărul de şiruri având $2 \cdot n$ paranteze care se închid corect.

Listing 14.6: Soluţia completă a problemei combinărilor

```
import java.io.*;
import io.Reader;

public class CombinariIt
{
  public static int combinariIter(int n, int k)
  {
    //...
  }

  public static void main(String args[])
  {
    System.out.print("n = ");
    int n = Reader.readInt();
    System.out.print("k = ");
    int k = Reader.readInt();

    if (n < 1 || k < 0 || k > n)
    {
      System.out.println("n trebuie sa fie >= 1");
      System.out.println("k trebuie sa fie >= 0 si" +
        " mai mic decat n");
      return;
    }

    System.out.println("Comb (" + n + ", " + k + ") = " +
      combinariIter(n, k));
  }
}
```

De exemplu, pentru $n = 2$, şirurile închise corect sunt $()()$ şi $(())$, deci răspunsul este 2.

Soluţia (celebră) a acestei probleme este reprezentată de aşa-numitele numere catalane (vezi [Cormen], pag. 261). Totuşi, datorită scopului acestui paragraf, vom da o altă soluţie a problemei, bazată pe o relaţie simplă de recurenţă.

Este cert că orice şir de paranteze închise corect începe cu o paranteză deschisă "(". Să considerăm acum paranteza care închide această primă paranteză. Ceea ce se află între aceste două paranteze este tot un şir de paranteze care se închid corect; la fel şi pentru şirul care se află în dreapta lor. Deci, un şir S de paranteze care se închid corect se poate scrie ca $(S1)S2$, unde $S1$ şi $S2$ sunt alte şiruri de paranteze care se închid corect, posibil vide.

Lungimea maximă a lui $S1$ este $2 \cdot n - 2$ (atinsă când $S2$ este vid), iar cea minimă este 0 (atinsă când $S2$ are lungimea $2 \cdot n - 2$). Să notăm cu $P(n)$

numărul de şiruri având $2n$ paranteze care se închid corect. Se observă cu uşurinţă că paranteza care închide prima paranteză poate fi pe oricare dintre poziţiile $2, 4, 6, \ldots, 2n$. Astfel, dacă paranteza este pe poziţia 2, atunci şirul $S1$ este vid, iar şirul $S2$ va avea lungimea $2n - 2 = 2(n-1)$, deci vor exista $P(0)P(n-1)$ şiruri de paranteze care se închid corect. Analog, dacă paranteza este pe poziţia 4, şirul $S1$ va avea lungimea 2, iar şirul $S2$ va avea lungimea $2n - 4 = 2(n-2)$, deci vom avea $P(1)P(n-2)$ şiruri care se închid corect. Obţinem astfel recurenţa:

$$P(n) = \begin{cases} 1 & pentru\, n = 0 \\ \sum_{k=0}^{n-1} P(k)P(n-k-1) & pentru\, n \geq 1 \end{cases}$$

Implementarea recurenţei de mai sus este simplă şi este prezentată în **Listing 14.7**.

Listing 14.7: Soluţia problemei parantezelor

```
import java.io.*;
import io.Reader;

public class Paranteze
{
  public static int parantezeIter(int n)
  {
    int[] p = new int[n + 1];
    p[0] = 1;

    for (int i = 1; i <= n; i++)
    {
      for (int k = 0; k <= i - 1; k++)
      {
        p[i] += p[k] * p[i - k - 1];
      }
    }

    return p[n];
  }

  public static void main(String args[])
  {
    System.out.print("n = ");
    int n  = Reader.readInt();

    if (n < 0) return;

    System.out.println("Numarul de siruri cu " + 2 * n +
      " paranteze care se inchid corect este: " +
      parantezeIter(n));
```

```
    }
}
```

Descopunerea unui număr natural ca sume distincte

Problema descompunerii unui număr natural în sume distincte este un alt exemplu în care recurenţa care furnizează soluţia trebuie construită de către programator. Enunţul problemei este:

> *Dându-se un număr natural n, se cere să se determine numărul de moduri distincte[3] în care acesta se poate scrie ca sumă de numere naturale nenule.*

De exemplu, numărul $n = 5$ se poate scrie ca:

$1 + 1 + 1 + 1 + 1$
$1 + 1 + 1 + 2$
$1 + 1 + 3$
$1 + 2 + 2$
$1 + 4$
$2 + 3$
5

Astfel, pentru $n = 5$ am obţinut 7 descompuneri (5 se consideră aici ca fiind sumă; se poate modifica problema excluzând această posibilitate, dar rezolvarea se păstrează, din numărul total de descompuneri trebuind doar să fie scăzut 1).

Pentru a obţine recurenţa pentru această problemă, vom rezolva mai întâi un caz particular:

> *Dându-se un număr natural n, să se determine numărul de moduri distincte în care acesta poate fi descompus în sumă de m numere naturale.*

Să notăm cu $S(n, m)$ numărul de moduri distincte în care n se poate scrie ca sumă de m termeni. Evident, soluţia problemei originale va fi

$$\sum_{i=1}^{n} S(n, i)$$

Problema originală se reduce aşadar la a calcula $S(n, i)$ $i = 1, 2, \ldots n$.

Se observă cu uşurinţă că $S(i, 1) = S(i, i) = 1$ (acestea sunt condiţiile iniţiale). De asemeni, $S(i, i + k) = 0$ pentru oricare k număr natural pozitiv.

[3]*Adunarea fiind comutativă, vom considera că descumpunerile $2 + 3$ şi $3 + 2$ sunt echivalente.*

Listing 14.8: Metoda de calcul a numărului de descompuneri

```
public static int descompuneNumar( int n )
{
 int s = new int[n][n] ; //alocam memorie pentru tabelul s
 //construim tabelul s
 for( int i=0; i<n; ++i )
  {
   s[i][0] = s[i][i] = 1 ; //conditiile initiale
   for( int j=1; j<i; ++i )
   {
     for( int k=0; k<j; ++k )
     {
       s[i][j] += s[i-j][k] ;
     }
   }
 }
 int nrDescompuneri = 0 ; //numarul de descompuneri ale lui n
 for( int i=0; i<n; ++i )
 {
   nrDescompuneri += s[n-1][i] ;
 }
 return nrDescompuneri ;
}
```

Putem determina o relaţie de recurenţă pentru $S(i, j)$ în cazul general ($i > j$) făcând următoarea observaţie simplă:

> Fiecare dintre cei j termeni care însumaţi dau i sunt mai mari sau egali cu 1. Dacă scădem valoarea 1 din fiecare dintre termeni, suma se va micşora cu j, iar termenii care fuseseră 1 devin 0, deci dispar.

Ţinând cont de această observaţie şi de faptul că numărul de termeni care sunt mai mari strict decât 1 este de minim 1 şi maxim j, putem scrie următoarea relaţie de recurenţă:

$$S(i,j) = \sum_{k=1}^{j} S(i-j,k)$$

Metoda `descompuneNumar()` din **Listing 14.8** rezolvă recurenţa de mai sus folosind trei cicluri `for` imbricate.

Analizând structura ciclurilor imbricate din metoda de mai sus, reiese că aceasta are o complexitate în $O(n^3)$. Putem însă reduce complexitatea la $O(n^2)$

folosind următorul artificiu de calcul:

$$S(i+1,j+1) = \sum_{k=1}^{j+1} S(i-j,k) = \sum_{k=1}^{j} S(i-j,k) + S(i-j,j+1) =$$

$$= S(i,j) + S(i-j,j+1)$$

Astfel, termenii $S(i,j)$ sunt calculaţi în $O(1)$ şi nu în $O(n)$, cum erau calculaţi înainte. Modificarea algoritmului pentru noua formulă este simplă şi o lăsăm ca exerciţiu.

În concluzie, deşi problemele prezentate în această secţiune nu sunt probleme de programare dinamcă propriu-zise, deoarece nu sunt de optimizare, ele constituie un excelent exerciţiu pentru înţelegerea principiilor programării dinamice, şi mai ales pentru deprinderea abilităţii de a găsi relaţii de recurenţă adecvate pentru exprimarea soluţiei.

14.3 Fundamentare teoretică

În paragraful anterior am făcut o scurtă trecere în revistă a unor probleme simple a căror rezolvare constă în determinarea şi calcularea unor recurenţe matematice. Am accentuat tratarea iterativă în contrast cu cea recursivă a relaţiilor de recurenţă, subliniind în acelaşi timp şi modul în care problemele se suprapun în cazul rezolvării recursive. Totuşi, aceste probleme nu pun în evidenţă decât un anumit aspect al programării dinamice, fără a aparţine propriu-zis acestui domeniu. În acest capitol vom urmări să vedem care sunt caracteristicile esenţiale ale problemelor de programare dinamică pe baza unui exemplu clasic: *problema triunghiului de numere.*

În cazul acestei probleme se dă un triunghi de numere z_{ij}, $j <= i$ având forma:

Figura 14.4: Triunghi de numere de dimensiune n

z_{11}
z_{21} z_{22}
...
z_{n1} z_{n2} $\ldots$ z_{nn}

Pentru $n = 4$, un posibil triunghi de numere este:

2
6 9
3 4 5
9 5 7 6

Un *drum* în acest triunghi este o secvenţă de n elemente ale triunghiului (n este numărul de linii) care începe întotdeauna cu elementul de pe prima linie, z_{11}, (2 în exemplul nostru) şi coboară linie cu linie până ajunge pe ultima linie a triunghiului. Din elementul z_{ij} se poate merge fie în $z_{i+1\,j}$, fie în $z_{i+1\,j+1}$. Elementele îngroşate din exemplul următor reprezintă un posibil drum:

2
6 9
3 **4** 5
9 5 **7** 6

Problema constă în a determina un drum în triunghi pentru care suma elementelor să fie maximă. În exemplul nostru, un drum de sumă maximă este:

2
6 **9**
3 4 **5**
9 5 **7** 6

având valoarea $2 + 9 + 5 + 7 = 23$.

Înainte de a ne avânta să rezolvăm o problemă folosind programarea dinamică, trebuie să ne convingem că nu putem folosi şi strategii mai simple (cum ar fi Greedy) pentru a obţine soluţia optimă. O strategie Greedy în cazul problemei triunghiului ar porni din vârful triungiului şi ar coborî la fiecare pas prin elementul mai mare. Aplicând această strategie pe exemplul anterior, am porni din 2, apoi am coborî în 9 (deoarece $9 > 6$), apoi am continua cu 5 respectiv 7. Am obţinut astfel drumul $2, 9, 5, 7$, care este chiar drumul optim! Am putea fi tentaţi să credem că putem aplica strategia Greedy. Să considerăm însă exemplul următor:

2
4 3
6 5 9

Soluţia obţinută prin strategia Greedy este $2, 4, 6$, având lungimea totală 12, în timp ce soluţia optimă este $2, 3, 9$, având lungimea totală 14. Rezultă aşadar că strategia Greedy nu generează întotdeauna soluţia optimă.

Pentru a rezolva o problemă printr-un algoritm de programare dinamică, o primă etapă esenţială constă în a **descompune (structura) problema în subprobleme asemănătoare ca structură cu problema iniţială**.

Structurarea pe subprobleme asemănătoare se face în cazul problemei triunghiului pe baza următoarei observaţii: fiecare element din triunghi poate fi

considerat vârful unui "sub-triunghi". De exemplu elementul 9 de pe poziţia $(2, 2)$ din triunghiul de mai sus este vârful sub-triunghiului

9
4 5
5 7 6

iar elementul 6 de pe poziţia $(2, 1)$ este vârful sub-triunghiului

6
3 4
9 5 7

Odată determinate subproblemele, începem prin a le rezolva subproblemele banale (de dimensiune minimă). Astfel, vom începe prin a determina un drum de sumă maximă pentru subproblemele generate de elementele din ultima linie a triunghiului. Acest lucru este banal, întrucât triunghiurile generate de către elementele de pe ultima linie au un singur element, deci drumul maxim este dat chiar de elementele respective:

9 5 7 6

Având rezolvate subproblemele generate de elementele de pe ultima linie, vom trece să rezolvăm subproblemele generate de elementele de pe penultima linie $(n-1)$. Pentru fiecare element de pe penultima linie avem două posibilităţi: fie se merge pe elementul de pe linia n aflat chiar dedesubt, fie pe elementul de pe linia n aflat în dreapta. Evident, vom alege varianta (subproblema) pentru care drumul corespunzător este mai mare. De exemplu, pentru elementul 3, vom prefera să mergem pe traseul $3 \rightarrow 9$ (3 + 9 = 12) decât pe traseul $3 \rightarrow 5$ (3 + 5 = 8). Astfel, valoarea drumului optim pentru subproblemele generate de elementele de pe penultima linie este:

12 11 12
9 8 7 6

Analog procedăm şi pentru elementele aflate pe a antepenultima linie (a doua în cazul nostru) linie. Şi în acest caz putem merge fie pe elementul aflat dedesubt fie pe elementul aflat în dreapta-jos. Vom alege, desigur, elementul a cărui subproblemă are valoarea mai mare. De exemplu, din elementul 6 am putea coborî fie prin 3, fie prin 4. Întrucât subproblema generată de 3 are drumul maxim de valoare 12, iar subproblema generată de 4 are drumul maxim de valoare 11, vom alege drumul care trece prin 3. Astfel, valoarea drumului optim pentru subproblemele generate de elementele de pe antepenultima linie este:

18 21
12 11 12
9 8 7 6

În final, pentru elementul din vârful triunghiului, vom alege să mergem pe traseul din dreapta, deoarece subproblema generată de elementul din dreapta-

jos (9) are valoare mai mare. Obţinem astfel valoarea drumului optim pentru elementul din vârf (care este de fapt valoarea optimă pentru problema originală), ca fiind $2 + 21 = 23$:

23
18 21
12 11 12
9 8 7 6

14.4 Principiul optimalităţii

Desigur că structurarea în subprobleme ar fi fost lipsită de sens dacă nu ne-ar fi permis să rezolvăm subproblemele corespunzătoare unui nivel bazându-ne pe soluţiile subproblemelor deja rezolvate. Am anticipat deci faptul că soluţia unei subprobleme se determină pe baza soluţiei subproblemelor de dimensiune mai mică, deja rezolvate. În această situaţie, vom spune că soluţia subproblemei x ***provine*** din soluţiile subproblemelor $y_1, y_2, \dots y_n$.

Problemele de programare dinamică sunt în general probleme de *optimizare*, iar problema triunghiului nu face nici ea excepţie. Din acest motiv, este necesar ca şi subproblemele în care am descompus problema originală să fie tot de optimizare. În fine, cea mai importantă caracteristică a problemelor de programare dinamică este următoarea:

> Dacă soluţia subproblemei x *provine* din soluţiile subproblemelor $y_1, y_2, \dots y_n$ şi soluţia subproblemei x este optimală, atunci şi soluţiile subproblemelor $y_1, y_2, \dots y_n$ sunt optimale.

Această proprietate poartă numele de ***principiul optimalităţii*** şi constituie piatra de temelie a programării dinamice.

În general, după ce am realizat o structurare a problemei iniţiale în subprobleme, trebuie verificat principiul optimalităţii. Această verificare se face cel mai adesea prin reducere la absurd.

Observaţie: Dificultatea nu constă în a verifica principiul optimalităţii, ci în a găsi o structurare a problemei în subprobleme care să verifice acest principiu.

Să demonstrăm acum faptul că problema triunghiului de numere respectă principiul optimalităţii. O soluţie a unei subprobleme constă într-un traseu optimal care porneşte din vârful triunghiului corespunzător subproblemei şi ajunge la baza triunghiului. Notăm acest traseu cu $a_k, a_{k+1}, \dots, a_n$, unde a_k este un element de pe linia k, a_{k+1} este un element de pe linia $k + 1$, iar a_n este un element de pe ultima linie. Pentru a se respecta principiul optimalităţii trebuie ca şi subtraseul $a_{k+1}, \dots, a_n$ să fie optimal pentru sub-triunghiul

generat de elementul a_{k+1}. Verificarea se face uşor prin reducere la absurd. Dacă admitem că ar exista un alt traseu care să pornească din a_{k+1}, notat cu $a_{k+1}, b_{k+2}, \ldots, b_n$, care să fie mai bun decât traseul $a_{k+1}, \ldots, a_n$, atunci traseul $a_k, a_{k+1}, b_{k+2} \ldots, b_n$ ar fi mai bun decât $a_k, a_{k+1}, \ldots, a_n$, ceea ce contrazice ipoteza că $a_k, a_{k+1}, \ldots, a_n$ este optim. Rezultă deci că structurarea în subprobleme realizată de noi respectă principiul optimalităţii.

Nu ne rămâne acum decât să scriem formula de recurenţă care descrie cum se obţine soluţia unei subprobleme pe baza subproblemelor de dimensiune mai mică deja rezolvate. Să notăm cu l_{ij} valoarea traseului optim pentru subproblema generată de elementul z_{ij}, aflat pe linia i şi coloana j ($i \geq j$). Este uşor de observat că $l_{ni} = z_{ni} \, \forall i = 1 \ldots n$, deoarece drumul maxim pentru elementele de pe ultima linie este reprezentat chiar de elementul respectiv. Am văzut că l_{ij} se obţine fie din $l_{i+1\,j}$ fie din $l_{i+1\,j+1}$ la care se adaugă valoarea elementului z_{ij}:

$$l_{ij} = z_{ij} + \max\{l_{i+1\,j}, l_{i+1,j+1}\} \;\; \forall i = 1 \ldots n-1, \; j = 1 \ldots i.$$

Valoarea lui l_{11} reprezintă chiar soluţia pentru problema originală.

Calcul recurenţei de mai sus este realizat de metoda `drumMaximTriunghi()` din **Listing 14.9**.

Reciproca principiului optimalităţii nu este adevărată

O altă observaţie importantă este aceea că, în general, reciproca principiului optimalităţii nu este adevărată. Cu alte cuvinte, combinând două sau mai multe sub-soluţii optime nu se obţine neapărat tot o soluţie optimă. Să presupunem că dorim să aflăm drumul de lungime minimă dintre două localităţi, să zicem Braşov şi Constanţa. Conform principiului optimalităţii, dacă avem un drum optim, x, între Braşov şi Constanţa care trece prin Ploieşti, atunci subdrumul lui x de la Ploieşti la Constanţa este de asemenea optim. Totuşi, dacă avem un drum optim între Braşov şi Suceava şi un alt drum optim între Suceava şi Constanţa, prin combinarea acestor două drumuri vom obţine un traseu Braşov-Suceava-Constanţa care este departe de a fi optim. Rezultă deci că reciproca principiului optimalităţii nu este adevărată în acest caz.

Este important să înţelegem că principiul optimalităţii nu ne asigură că prin combinarea soluţiilor optime se obţine tot o soluţie optimă. În schimb, el ne asigură că *în căutarea unei soluţii optime nu trebuie să luăm în considerare decât subsoluţiile optime*.

Listing 14.9: Rezolvarea recurenţei pentru problema triunghiului

```
/**
 * Calculeaza formula de recurenta pentru matricea L.
 * @return Lungimea drumului maximal pentru triunghi
 * care se presupun a fi membri ai clasei care contine metoda.
 */
public static int drumMaximTriunghi()
{
  //initializeaza ultima linie din L
  for (int i = 0; i < z.length; ++i)
  {
    l[z.length - 1][i] = z[z.length - 1][i];
  }

  //calculeaza elementele din l linie cu linie
  for (int i = z.length - 2; i >= 0; --i)
  {
    for (int j = 0; j <= i; ++j)
    {
      l[i][j] = z[i][j] + Math.max(l[i+1][j], l[i+1][j+1]);
    }
  }

  return l[0][0];
}
```

O altă descompunere în subprobleme

Este interesant de remarcat faptul că problema triunghiului de numere admite şi o altă descompunere în subprobleme care verifică şi ea principiul optimalităţii, şi care conduce la o altă soluţie a problemei. În descompunerea în subprobleme pe care am dat-o anterior am notat prin l_{ij} lungimea drumului maxim care începe cu elementul z_{ij}. De data aceasta, vom nota cu m_{ij} lungimea drumului maxim care se *încheie* cu elementul z_{ij}. Considerând acelaşi triunghi de numere:

2
6 9
3 4 5
9 5 7 6

subproblema corespunzătoare elementului 4 din linia a treia este dată de "tăietura" în sus generată de acest element:

2
6 **9**
3 **4**

unde drumul maxim care se termină cu 4 a fost îngroşat. Verificarea principiului optimalităţii se face foarte uşor şi de această dată: este clar că un drum de lungime maximă este format din sub-drumuri care sunt tot de lungime maximă.

Odată determinate subproblemele, începem din nou prin a le rezolva pe cele triviale. Astfel, vom determina mai întâi un drum de sumă maximă pentru elementul din prima linie a triunghiului. Întrucât triunghiul generat de către elementul din prima linie are un singur element, drumul optim este dat chiar de elementul respectiv.

Având rezolvate subproblemele generate de elementele de pe prima linie, vom trece să rezolvăm subproblemele generate de elementele de pe umătoarele linii, obţinând triunghiul:

2
8 **11**
11 15 **16**
20 20 **23** 22

Pentru fiecare element al triunghiului (mai puţin cele din prima şi ultima coloană a fiecărei linii) avem două posibilităţi: fie se ajunge la el de pe elementul aflat imediat deasupra, fie de pe elementul aflat pe diagonala stânga-sus. Evident, vom alege traseul (subproblema) pentru care drumul corespunzător este mai mare. De exemplu, pentru elementul 4, vom prefera să mergem pe traseul ... $\to 9 \to 4$ (având lungimea $11+4=15$) decât pe traseul ... $\to 6 \to 4$ (având lungimea $8+4=12$).

Soluţia subproblemei iniţiale este dată de subproblema corespunzătoare elementului de pe ultima linie, a cărei valoare este maximă. Rezultă deci că lungimea drumului maxim este 23, aşadar am regăsit valoarea obţinută aplicând cealaltă descompunere în subprobleme. Formulele de recurenţă pentru calculul elementelor m_{ij} sunt:

$$m_{ij} = z_{ij} + \max(m_{i-1\,j-1}, m_{i-1\,j}) \qquad m_{11} = z_{11},$$

unde facem convenţia că $m_{0\,i} = m_{i+1\,i} = 0$.

14.4.1 Metoda "înainte" şi metoda "înapoi"

Aşa cum probabil aţi remarcat deja, cele două descompuneri în subprobleme pe care le-am descris pentru problema triunghiului de numere sunt complementare:

- în prima variantă se consideră sub-drumuri care încep cu un anumit element, în cea de-a doua variantă se consideră sub-drumuri care se termină cu un anumit element;

- în prima variantă se construieşte drumul optim pornind de la baza triunghiului, în timp ce în cea de-a doua variantă se construieşte drumul optim pornind de la vârful triunghiului;

- în prima variantă recurenţa merge "înainte" de la elementul i la elementul $i+1$, în cea de-a doua variantă recurenţa merge "înapoi", de la elementul i la elementul $i-1$.

Unii autori clasifică metodele de rezolvare a problemelor de programare dinamică în trei clase:

1. Metoda "înainte", în care ne folosim de faptul că optimalitatea subsoluţiei $a_k, a_{k+1}, \dots a_n$ implică optimalitatea subsoluţiei $a_{k+1}, \dots, a_n$;

2. Metoda "înapoi", în care ne folosim de faptul că optimalitatea subsoluţiei $a_1, \dots a_{k-1}, a_k$ implică optimalitatea subsoluţiei $a_1, \dots a_{k-1}$;

3. Metoda "mixtă", în care ne folosim de faptul că optimalitatea soluţiei de tipul $a_1, \dots, a_k, \dots a_n$ implică optimalitatea atât a subsoluţiei $a_1, \dots a_{k-1}$ cât şi a sub-soluţiei $a_{k+1}, \dots, a_n$.

În prima descompunere în subprobleme am aplicat metoda "înainte", iar în a doua descompunere în subprobleme am aplicat metoda "înapoi". Vom prezenta în paragraful 14.5 şi un exemplu de problemă (parantezarea unui şir de matrice) în care se aplică metoda "mixtă".

14.4.2 Determinarea efectivă a soluţiei optime

Deşi rezolvarea recurenţelor asociate unei probleme de programare dinamică ne conduce la aflarea valorii optime căutate, ea nu furnizează şi soluţia efectivă pentru care se obţine acea valoare optimă. De exemplu, în cazul problemei triunghiului de numere am calculat care este valoarea drumului optimal¸ dar nu am găsit efectiv care este drumul pentru care se obţine valoarea optimă.

Vestea bună este că aflarea soluţiei optime a problemei este în general simplă şi se bazează de cele mai multe ori pe reconstituirea "traseului" prin care s-a obţinut valoarea soluţiei optime. În cazul problemei triunghiului de numere putem afla cu uşurinţă care este drumul pentru care se obţine soluţia optimă,

reconstituind traseul prin care s-a obţinut valoarea optimă. Pentru exemplul nostru, triunghiul construit prin rezolvarea relaţiilor de recurenţă era:

23			
18	**21**		
12	11	**12**	
9	5	**7**	6

Se observă cu uşurinţă că valoarea 23 din vârful triunghiului a fost obţinută din elementul 21 aflat în dreapta jos (prin adăugarea valorii 2 din vârful triunghiului original), valoarea 21 a fost obţinută din elementul 12 aflat la dreapta, iar 12 din 7.

Metoda `afiseazaDrumMaxim()` din **Listing 14.10** calculează drumul maxim pentru problema triunghiului. La linia 33 se iniţializează variabila `s` cu valoarea drumului maxim, aflată în vârful triungiului. Drumul propriu-zis este construit în ciclul `while` din liniile 38-53, în care se parcurge matricea `l` de sus în jos, linie cu linie. La fiecare iteraţie, variabila `s` conţine valoarea către care trebuie să coborâm. Astfel, dacă `s` este egală cu `l[i+1][j+1]` (linia 46) se coboară direct în `l[i+1][j+1]`, altfel se coboară în `l[i+1][j+2]`.

Listing 14.10: Soluţia completă a problemei drumului maxim

```
1 import java.io.*;
2 import io.Reader;
3
4 public class DrumulMaxim
5 {
6   public static int drumMaximTriunghi(int[][] z)
7   {
8     int[][] l = new int[z.length][z.length];
9
10    for (int i = 0; i < z.length; i++)
11    {
12      l[z.length - 1][i] = z[z.length - 1][i];
13    }
14
15    for (int i = z.length - 2; i >= 0; i--)
16    {
17      for (int j = 0; j <= i; j++)
18      {
19        l[i][j] = z[i][j] +
20          Math.max(l[i + 1][j],
21              l[i + 1][j + 1]);
22      }
23    }
24
25    afiseazaDrumMaxim(z, l);
26
27    return l[0][0];
```

```
  }

  public static void afiseazaDrumMaxim(int[][] z, int[][] l)
  {
    int s = l[0][0];
    int i = 0;
    int j = 0;

    System.out.print("Drumul maxim: ");
    while (s != 0)
    {
      System.out.print(z[i][j] + " ");

      s -= z[i][j];

      if (i + 1 < z.length)
      {
        if (s == l[i + 1][j + 1])
        {
          j++;
        }

        i++;
      }
    }
    System.out.println();
  }

  public static void main(String args[])
  {
    System.out.print("Dimensiune triunghi: ");
    int n  = Reader.readInt();

    int[][] z = new int[n][n];
    System.out.println("Elementele triunghiului:");
    for (int i = 0; i < n; i++)
    {
      for (int j = 0; j <= i; j++)
      {
        System.out.print("z[" + i + "][" + j
          + "] = ");

        z[i][j] = Reader.readInt();
      }
    }

    System.out.println("Suma maxima = " + drumMaximTriunghi(z));
  }
}
```

14.5 Înmulţirea unui şir de matrice

Următorul exemplu de problemă de programare dinamică pe care îl vom studia este problema înmulţirii optimale a unui şir de matrice. În cazul acestei probleme se dă o secvenţă $A_1, A_2 \ldots, A_n$ de matrice care trebuie înmulţite. Cu alte cuvinte, se doreşte calcularea produsului

$$A_1 A_2 \ldots A_n,$$

unde A_i are dimensiunile $d_{i-1} \times d_i$. Având în vedere faptul că înmulţirea matricelor este asociativă, calculul produsului de mai sus se poate face în mai multe moduri, în funcţie de ordinea în care decidem să realizăm operaţiile de înmulţire. De exemplu, pentru $n = 3$ există două moduri în care se poate calcula produsul:

$$A_1(A_2 A_3)$$

sau

$$(A_1 A_2) A_3.$$

Unii cititori îşi pot pune în mod legitim întrebarea: *Ţinând cont de faptul că înmulţirea matricelor este asociativă, oricum am pune parantezele, rezultatul final al calcului va fi acelaşi. Aşadar ce sens are să ne preocupăm de ordinea de realizare a operaţiilor de înmulţire?* Răspunsul este că numărul de operaţii elementare de înmulţire este diferit, funcţie de modul în care alegem să punem parantezele. De exemplu, pentru $n = 3$, să presupunem că cele 3 matrice au respectiv dimensiunile $(10, 50)$, $(50, 20)$ şi $(20, 1)$. Numărul de operaţii necesar pentru a înmulţi două matrice de dimensiune (m, n) şi (n, p) este $m \cdot n \cdot p$, după cum reiese clar din algoritmul din **Listing 14.11** (considerăm ca barometru operaţia de înmulţire scalară $a[i][k] * b[k][j]$). Dacă vom calcula produsul celor 3 matrice după primul mod $(A_1(A_2 A_3))$, vom face $50 \times 20 \times 1 = 1000$ de operaţii pentru a calcula produsul $A_2 A_3$, plus încă $10 \times 50 \times 1 = 500$ de operaţii pentru a înmulţi rezultatul cu A_1. Aşadar, în total vom realiza $1000 + 500 = 1500$ de înmulţiri scalare. Dacă vom calcula produsul celor 3 matrice în al doilea mod $((A_1 A_2) A_3)$, vom face $10 \times 50 \times 20 = 10000$ de operaţii pentru a calcula produsul $A_1 A_2$ plus încă $10 \times 20 \times 1 = 400$ operaţii pentru a calcula produsul rezultatului cu A_3. Vom face aşadar un total de 10400 de operaţii ceea ce înseamnă cam de 7 ori mai mult decât în varianta precedentă. În concluzie, are sens să ne punem problema de a găsi cea mai eficientă metodă de a realiza această înmulţire.

O soluţie imediată a problemei ar fi să se găsească toate modurile posibile de parantezare a şirului de matrice şi să se aleagă cea pentru care numărul de

operaţii este minim. Totuşi, numărul de parantezări pentru un şir de lungime n este (vezi [Cormen], p. 261)

$$\frac{1}{n}C_{2n-2}^{n-1} \in \Omega(4^n/n^{3/2})$$

ceea ce exclude din start posibilitatea căutării exhaustive datorită numărului exponenţial de variante.

Listing 14.11: Înmulţirea a două matrice de dimensiune m-n respectiv n-p. Se observă cu uşurinţă că numărul de operaţii de înmulţire realizate este m*n*p

```
/**
 * Calculeaza produsul a doua matrice de numere intregi.
 * @return O matrice de numere intregi care reprezinta produsul
 * matricelor date ca parametru.
 */
public static int[][] produs(int[][] a, int[][] b)
{
  int[][] produs = new int[a.length][b[0].length];
  for (int i = 0; i < a.length; ++i)
  {
    for (int j = 0; j < b[0].length; ++j)
    {
      produs[i][j] = 0;
      for (int k = 0; k < b.length; ++k)
      {
        produs[i][j] += a[i][k]*b[k][j];
      }
    }
  }

  return produs;
}
```

Caracterizarea substructurii optime

O parantezare optimă a produsului $A_1A_2\ldots A_n$ împarte secvenţa între A_k şi A_{k+1}, unde k este un număr natural în intervalul $1..n-1$. Aceasta înseamnă că mai întâi se calculează $A_1\ldots A_k$, apoi $A_{k+1}\ldots A_n$ şi în final se înmulţesc cele două rezultate pentru a obţine produsul $A_1A_2\ldots A_n$. Numărul total de înmulţiri realizate este egal cu numărul de înmulţiri necesare pentru calculul lui $A_1\ldots A_k$ plus numărul de înmulţiri necesare pentru a calcula $A_{k+1}\ldots A_n$ la care se adaugă numărul de operaţii necesar pentru a înmulţi cele două rezultate.

Această modalitate de împărţire ne duce cu gândul la a considera subprobleme de forma $P_{ij} = A_i A_{i+1} \ldots A_j$, cu $1 \leq i \leq j \leq n$. Observaţia esenţială în cazul acestei probleme este că dacă parantezarea optimă pentru a calcula $A_i \ldots A_j$ împarte produsul pe poziţia k,

$$P_{ij} = (A_i \ldots A_k)(A_{k+1} \ldots A_j)$$

atunci parantezarea subşirului $A_i \ldots A_k$ din cadrul parantezării optime a şirului $A_i \ldots A_j$ este o parantezare optimă pentru şirul $A_i \ldots A_k$ (dacă nu ar fi optimă, atunci înlocuirea respectivei parantezări cu cea optimă ar produce o altă parantezare pentru $A_i \ldots A_j$ al cărei cost ar fi mai mic decât cel optim, ceea ce este absurd). O observaţie similară este valabilă şi pentru parantezarea lui $A_{k+1} \ldots A_j$ din cadrul parantezării optime a lui $A_i \ldots A_j$, care trebuie să fie optimă pentru şirul $A_{k+1} \ldots A_j$. Am arătat aşadar că o soluţie optimă pentru problema noastră este compusă din subsoluţii optime ale subproblemelor, ceea ce ne permite să aplicăm programarea dinamică.

Obţinerea formulelor de recurenţă

Odată găsită o descompunere în subprobleme care respectă principiul optimalităţii, putem trece la definirea valorii unei soluţii optime în funcţie de soluţiile optime ale subproblemelor. Fie m_{ij} numărul minim de înmulţiri necesare pentru a calcula $P_{ij} = A_i \ldots A_j$. Scopul final este să calculăm m_{1n}. Dacă $i = j$, atunci şirul constă într-o singură matrice, deci nu avem nevoie de nici o înmulţire scalară. Vom avea astfel

$$m_{ii} = 0, \forall i = 1..n.$$

Pentru a calcula m_{ij} în cazul general ($i < j$) ne vom folosi de substructura optimală definită în paragraful anterior. Să presupunem că o parantezare optimă a şirului $A_i \ldots A_j$ împarte produsul între A_k şi A_{k+1} cu $i \leq k < j$. Am arătat deja că în această situaţie numărul de operaţii necesar pentru a calcula P_{ij} este egal cu numărul de operaţii necesare pentru a calcula P_{ik} plus numărul de operaţii necesare pentru a calcula P_{k+1j} plus costul înmulţirii celor două matrice rezultat. Având în vedere faptul că P_{ik} are dimensiunea (d_{i-1}, d_k), iar P_{k+1j} are dimensiunea (d_k, d_j), evaluarea produsului $P_{ik}P_{k+1j}$ necesită $d_{i-1}d_k d_j$ operaţii. Obţinem astfel numărul de operaţii necesare pentru a calcula m_{ij}:

$$m_{ij} = m_{ik} + m_{k+1j} + d_{k-1}d_k d_j.$$

Ecuaţia de mai sus este valabilă în situaţia în care cunoaştem poziţia k în care se descompune produsul. Cum noi nu ştim care este această poziţie, vom încerca

toate variantele posibile pentru k şi o vom alege pe cea pentru care se obţine valoarea minimă. Din fericire, există doar $j - i$ valori posibile pentru k, şi anume $k = i, i+1, \dots j-1$. În consecinţă, formula recurentă pentru definirea costului minim a parentezării produsului $A_i \dots A_j$ devine

$$m_{ij} = \begin{cases} 0 & daca\, i = j \\ \min_{i \le k < j}\{m_{ik} + m_{k+1j} + d_{i-1}d_k d_j\} & daca\, i < j \end{cases} \quad (14.1)$$

Rezolvarea recurenţei şi calculul costului optimal

În acest moment este uşor să scriem un algoritm care să calculeze recurenţa pentru m_{ij}. O primă alternativă, destul de tentantă, este aceea de a implementa calculul recurenţei direct, folosind un algoritm recursiv. O analiză similară cu cea de la calculul recursiv al şirului lui Fibonacci (paragraful 14.2.1), ne convinge că această modalitate de calcul generează o complexitate exponenţială, cu nimic mai bună decât încercarea tuturor parantezărilor posibile.

În locul calculului recursiv al lui m_{ij} vom folosi din nou un tabel în care vom reţine valorile intermediare ale subproblemelor. Ca de obicei, calculul se va face *bottom-up*, adică se rezolvă mai întâi subproblemele de dimensiune mică, urmate apoi succesiv de subprobleme de dimensiune mai mare, până când se ajunge la problema iniţială. Metoda `sirMatrice()` din **Listing 14.12** implementează această strategie.

`sirMatrice()` calculează matricea m mergând pe probleme de dimensiune din ce în ce mai mare. La prima iteraţie ($l = 0$) algoritmul calculează $m[i][i+1]$ (costul şirurilor de lungime 2) pentru $i = 0 \dots n-2$ (indicii sunt deplasaţi faţă de notaţia anterioară deoarece în Java indexarea tablourilor începe de la 0). La a doua trecere prin ciclu ($l = 1$) se calculează $m[i][i+2]$ pentru $i = 0, 1, \dots, n-3$ (costul şirurilor de lungime 3) etc. Aşadar, elementele matricei m sunt calculate pe diagonală, mergând paralel cu diagonala principală, până când se ajunge în colţul din dreapta sus al matricei. **Figura** 14.5 ilustrează această procedură pentru un şir de $n = 6$ matrice, având dimensiunile $(20 \times 15), (15 \times 30), (30 \times 5), (5 \times 25), (25 \times 10), (10 \times 35)$.

Construirea efectivă a unei soluţii optime

Şi în cazul acestei probleme, rezolvarea recurenţei implică găsirea valorii optime, fără a da însă şi parantezarea efectivă pentru care s-a obţinut valoarea optimă. Ca de obicei, calculul soluţiei optime se face simplu, folosindu-ne de rezolvarea recurenţei. Pentru a putea găsi care este soluţia optimă, este suficient ca pentru fiecare produs $A_i \dots A_j$ să ştim care este poziţia k la care se împarte

Listing 14.12: Calculul recurenţei pentru înmulţirea şirului de matrice folosind un tabel m în care se reţin valorile subproblemelor

```
1 /**
2  * Calculeaza formula de recurenta pentru matricea m.
3  * Ne folosim de faptul ca elementele matricei l sunt
4  * initializate implicit cu 0.
5  * @return Costul optim pentru inmultirea matricelor a caror
6  * dimensiune este retinuta in sirul d care se considera a
7  * fi membru al clasei care contine metoda.
8  */
9 public static int sirMatrice()
10 {
11   int n = d.length - 1;
12   int[][] m = new int[n][n];
13   for (int l = 0; l < n - 1; ++l) //l este lungimea subsirului
14   {
15     for (int i = 0; i < n - l - 1; ++i)
16     {
17       j = i + l + 1;
18       m[i][j] = Integer.MAX_VALUE;
19       int temp;
20
21       for (int k = i; k < j; ++k)
22       {
23         temp = m[i][k] + m[k+1][j] + d[i]*d[k+1]*d[j+1];
24         if (temp < m[i][j])
25         {
26           m[i][j] = temp;
27         }
28       }
29     }
30   }
31
32   return m[0][n - 1];
33 }
```

în două subproduse. Aceasta implică să construim o matrice poz_{ij} care reţine poziţia în care şirul $A_i \dots A_j$ este împărţit în subproduse. Calculul lui poz_{ij} se face simplu, adăugând atribuirea

```
poz[i][j] = k;
```

după linia 26 din metoda `sirMatrice()` (**Listing 14.12**). Având la dispoziţie şirul *poz*, găsirea soluţiei optime se face simplu folosind metoda recursivă `sirMatriceSolutie()` din **Listing 14.13**.

Figura 14.5: Tabelul m rezultat în urma rezolvării recurenţei pentru problema înmulţirii şirului de matrice, pentru un şir de 6 matrice având dimensiunile $(20 \times 15), (15 \times 30), (30 \times 5), (5 \times 25), (25 \times 10), (10 \times 35)$

0	9000	3750	6250	6000	13000
	0	2250	5625	7250	12500
		0	3750	2750	13250
			0	1250	10500
				0	8750
					0

Listing 14.13: Construirea efectivă a soluţiei optime pentru problema înmulţirii şirului de matrice

```
/**
 * Metoda care determina efectiv produsul sirului de
 * matrice in mod optim bazandu-se pe matricea  calculata
 * de metoda sirMatrice.
 * i si j reprezinta limitele intre care se calculeaza produsul
 * @return String ce reprezinta modul de parantezare al matricelor
 */
public static String sirMatriceSolutie(int[][] poz, int i, int j)
{
  if (i < j) //sirul contine cel putin doua matrice
  {
    String s1 = sirMatriceSolutie(poz, i, poz[i][j]);
    String s2 = sirMatriceSolutie(poz, poz[i][j]+1, j);

   return "(" + s1 + "*" + s2 + ")" ;

  }
  else //sirul contine o singura matrice
  {
    return "A" + String.valueOf(i) ;
  }
}
```

Clasa `Matrice` din **Listing 14.14** afişează parantezarea optimă şi numărul de înmulţiri minim necesar pentru un şir de matrice ale cărui dimensiuni sunt citite de la tastatură.

Listing 14.14: Soluţia completă a problemei şirului de matrice

```
import java.io.*;
```

```
import io.Reader;

public class Matrice
{
  public static int sirMatrice(int[] d)
  {
    int n = d.length - 1;
    int[][] m = new int[n][n];
    int[][] poz = new int[n][n];

    for (int l = 0; l < n - 1; l++)
    {
      for (int i = 0; i < n - l - 1; i++)
      {
        int j = i + l + 1;
        m[i][j] = Integer.MAX_VALUE;
        int temp;
        for (int k = i; k < j; k++)
        {
          temp = m[i][k] + m[k + 1][j] +
            d[i] * d[k + 1] * d[j + 1];

          if (temp < m[i][j])
          {
            m[i][j] = temp;
            poz[i][j] = k;
          }
        }
      }
    }


    System.out.print("Ordinea inmultirii matricilor: " +
        sirMatriceSolutie(poz, 0, n - 1));
    System.out.println();

    return m[0][n - 1];
  }

  public static String sirMatriceSolutie(int[][] poz,
          int i, int j)
  {
    if (i < j)
    {
      String s1 = sirMatriceSolutie(poz, i, poz[i][j]);
      String s2 = sirMatriceSolutie(poz, poz[i][j] + 1, j);

      return "(" + s1 + " * " + s2 + ")";
    }
    else
```

```
    {
      return "A" + String.valueOf(i);
    }
  }

  public static void main(String[] args)
  {
    System.out.print("Numarul de matrice: ");
    int n  = Reader.readInt();

    if (n < 1) return;

    int[] d = new int[n + 1];

    System.out.println("Sirul dimensiunilor: ");

    for (int i = 0; i < n + 1; i++)
    {
      System.out.print("d[" + i + "]=");
      d[i]  = Reader.readInt();
    }

    System.out.println("Numarul minim de operatii necesar " +
      "inmultirii matricilor: " + sirMatrice(d));
  }
}
```

14.6 Subşir comun de lungime maximă

Următoarea problemă clasică de programare dinamică pe care o vom analiza este problema subşirului comun de lungime maximă. Un subşir al unui şir $x = x_1x_2\dots x_n$ se obţine prin eliminarea din şirul iniţial x a unor componente $x_{i_1}, x_{i_2}, \dots, x_{i_k}$; componentele care se elimină nu trebuie să fie neapărat consecutive. De exemplu, şirul aeg este un subşir al lui $abcdefgh$. Astfel, dându-se două şiruri, x şi y, spunem că şirul z este un *subşir comun* pentru x şi y dacă z este un subşir atât pentru x cât şi pentru y.

În problema subşirului comun de lungime maximă se dau două şiruri $x = x_1x_2\dots x_m$ şi $y = y_1y_2\dots y_n$ arbitrare şi se cere să se găsească un subşir *comun* al lor $z = z_1z_2\dots z_p$ (evident, $p <= \min(m, n)$) care să aibă lungimea maximă.

Exemplu: Pentru şirurile *dinamic* şi *inadecvat* subşirul comun de lungime maximă este $inac$, deoarece nu există nici un subşir comun de lungime mai mare decât 4. Este important de observat că subşirul comun de lungime maximă nu este neapărat unic. Astfel, pentru şirurile *abeiro* şi *bueor* un subşir

comun de lungime maximă este ber, iar altul este beo.

O abordare brutală a acestei probleme constă în generarea tuturor subşirurilor lui x şi alegerea celui mai lung care este în acelaşi timp şi subşir al lui y. Desigur, această soluţie nu este deloc eficientă, având în vedere faptul că există 2^m subşiruri ale lui x, deci timpul necesar pentru rezolvarea problemei ar fi exponenţial.

Caracterizarea substructurii optime (verificarea principiului optimalităţii)

Din fericire, această problemă se pretează la o structurare pe subprobleme care verifică principiul optimalităţii. Totuşi, în acest caz, structurarea în subprobleme care să verifice principiul optimalităţii nu este evidentă.

Subproblemele se definesc în mod natural pe baza unor perechi de "prefixe" ale celor două şiruri de intrare. Mai exact, dându-se un şir $x = x_1x_2 \ldots x_m$, un prefix al său este şirul $x_1x_2 \ldots x_i$, cu $i = 0, 1, \ldots m$. Astfel, pentru fiecare pereche de prefixe, $x = x_1 \ldots x_i, y = y_1 \ldots y_j$ ale şirurilor iniţiale vom calcula lungimea subşirului maximal comun, ajungând în final să rezolvăm şi problema noastră.

Pentru a verifica principiul optimalităţii vom enunţa trei propoziţii simple, în care plecăm de la ipoteza că $z_1z_2 \ldots z_p$ este un subşir maximal pentru $x_1x_2 \ldots x_m$ şi $y_1y_2 \ldots y_n$.

1. Dacă $x_m = y_n$, atunci $z_1z_2 \ldots z_{p-1}$ este un subşir comun maximal pentru $x_1x_2 \ldots x_{m-1}$ şi $y_1y_2 \ldots y_{n-1}$;

2. Dacă $x_m \neq y_n$ şi $z_p \neq x_m$, atunci $z_1z_2 \ldots z_p$ este un subşir comun maximal pentru $x_1x_2 \ldots x_{m-1}$ şi $y_1y_2 \ldots y_n$;

3. Dacă $x_m \neq y_n$ şi $z_p \neq y_n$, atunci $z_1z_2 \ldots z_p$ este un subşir comun maximal pentru $x_1x_2 \ldots x_m$ şi $y_1y_2 \ldots y_{n-1}$.

Demonstraţia celor trei afirmaţii de mai sus se face în mod banal prin reducere la absurd şi o lăsăm ca exerciţiu.

Caracterizarea din propoziţiile anterioare arată că un subşir comun maximal pentru două şiruri conţine un subşir comun maximal pentru prefixele celor două şiruri, deci problema are proprietatea de substructură optimă. Tot propoziţiile de mai sus furnizează şi formula de recurenţă care rezolvă problema.

Vom nota cu $l[i, j]$ lungimea subşirului comun maximal pentru prefixele $x_1 \ldots x_i$ şi $y_1 \ldots y_j$.

Ca şi condiţii iniţiale avem:

$$l[0, j] = 0, \ \forall j = 0, 1, \ldots n$$

$$l[i, 0] = 0, \ \forall i = 0, 1, \ldots m$$

deoarece atunci când unul dintre şiruri este vid (are lungime 0), subşirul comun maximal va fi întotdeauna vid.

Din cele trei propoziţii rezultă în mod clar faptul că având un subşir comun maximal pentru două şiruri oarecare $x_1x_2\dots x_m$ şi $y_1y_2\dots y_n$, acesta se va regăsi în subşirul comun maximal pentru una din variantele

$x_1x_2\dots x_{m-1}$ şi $y_1y_2\dots y_{n-1}$ dacă $x_m = y_n$,

sau

$x_1x_2\dots x_{m-1}$ şi $y_1y_2\dots y_n$ ori $x_1x_2\dots x_m$ şi $y_1y_2\dots y_{n-1}$ dacă $x_m \neq y_n$.

De aici reiese că pentru a calcula $l[m,n]$ sunt necesare valorile $l[m-1,n-1]$, $l[m-1,n]$ şi $l[m,n-1]$. Totuşi, cele trei propoziţii nu dau indicaţii exacte despre formula de recurenţă decât într-un singur caz, şi anume când $x_m = y_n$ (noi nu ştim ce valoare are z_p atunci când construim soluţia, deci nu putem distinge între variantele 2 şi 3). În situaţia în care $x_m \neq y_n$ putem alege fie $l[m-1,n]$, fie $l[m,n-1]$. Dintre aceste valori o vom alege evident pe cea mai mare, deoarece ea îndeplineşte proprietatea de optimalitate (se poate arăta acest lucru uşor, prin reducere la absurd). Astfel, vom alege

$$l[m,n] = max\{l[m-1,n], l[m,n-1]\}\ x_m \neq y_n$$

Aşadar, formula de recurenţă completă este:

$$l[i,\,j] = \begin{cases} 0 & ptr\, i = 0\, sau\, j = 0, \\ l[i-1,\,j-1]+1 & ptr\, i = 1..m, j = 1..n\, si\, x_i = x_j, \\ \max(l[i-1,\,j], l[i,\,j-1]) & ptr\, i = 1..m, j = 1..n\, si\, x_i \neq x_j. \end{cases}$$

Calculul recurenţei de mai sus se face simplu, folosind metoda `subsirComunMaximal()` din **Listing 14.15**.

Listing 14.15: Rezolvarea recurenţei pentru subşirul comun de lungime maximă

```
/**
 * Calculeaza formula de recurenta pentru matricea l.
 * Ne folosim de faptul ca elementele matricei l sunt
 * initializate implicit cu 0.
 * @return Lungimea subisrului comun maximal pentru x si y
 * care se presupun a fi membri ai clasei care contine metoda.
 */
public static int subsirComunMaximal()
{
    int m = x.length(); //x este String
    int n = y.length(); //y este String
```

```
  for (int i = 0; i < m; ++i)
  {
    for (int j = 0; j < n; ++j)
    {
      if (x.charAt(i) == y.charAt(j))
      {
        l[i+1][j+1] = l[i][j] + 1;
      }
      else
      {
        l[i+1][j+1] = Math.max(l[i][j+1], l[i+1][j]);
      }
    }
  }

  return l[m][n];

}
```

Am rezolvat astfel problema determinării lungimii subşirului comun maximal, dar, ca de obicei, nu am găsit subşirul efectiv care are lungimea maximă. Determinarea subşirului efectiv se poate realiza uşor pe baza matricei l calculată în metoda `subsirComunMaximal()` din **Listing 14.15**, urmărind care a fost "traseul" care a condus la subşirul de lungime maximă (dat de $l[m,n]$). Am evidenţiat deja faptul că $l[i,j]$ poate proveni doar dintr-unul din elementele $l[i-1,j-1]$, $l[i-1,j]$, sau $l[i,j-1]$. Începem cu elementul $l[m,n]$ al matricei; acesta provine din $l[m-1,n-1]$ dacă $x_m = y_n$ sau din maximul dintre $l[m-1,n]$ şi $l[m,n-1]$ în caz contrar. Mergem astfel pas cu pas până când ajungem la $l[0,0]$. Din drumul de la $l[m,n]$ la $l[0,0]$ construit anterior reţinem doar elementele $l[i,j]$ pentru care $x_i = y_j$. Afişând în ordine inversă aceste elemente, obţinem chiar şirul cerut (**Listing 14.16**).

Listing 14.16: Determinarea efectivă a subşirului comun maximal

```
/**
 * Determina care este subsirul comun maximal pe baza
 * matricei l si a sirurilor x si y (care se presupun a fi
 * membri ai clasei.
 * @return Un String care reprezinta subsirul comun maximal.
 */
public static String determinaSubsir(int i, int j)
{
  if (i != 0 && j != 0)
  {
    if (x.charAt(i - 1) == y.charAt(j - 1))
```

```
    {
      return determinaSubsir(i - 1, j - 1) + x.charAt(i - 1);
    }
    else
    {
      if (l[i][j] == l[i-1][j])
      {
        return determinaSubsir(i - 1, j);
      }
      else
      {
        return determinaSubsir(i, j - 1);
      }
    }
  }
  else
  {
    return "";
  }
}
```

Clasa `SubsirComun` din **Listing 14.17** citeşte două stringuri de la tastatură, după care afişează subşirul lor comun de lungime maximă.

Listing 14.17: Soluţia completă a problemei subşirului comun maximal

```
import java.io.*;
import io.Reader;

public class SubsirComun
{
  private static String x;
  private static String y;
  private static int[][] l;

  public static int subsirComunMaximal()
  {
    //...
    System.out.println("Subsirul comun maximal: " +
      determinaSubsir(m, n));

    return l[m][n];
  }

  public static String determinaSubsir(int i, int j)
  {
    //...
  }

  public static void main(String args[])
  {
```

```
    System.out.print("x = ");
    x = Reader.readString();

    System.out.print("y = ");
    y = Reader.readString();

    System.out.println("Lungimea subsirului comun maximal: "
      + subsirComunMaximal());
  }
}
```

14.7 Distanţa Levensthein

O problemă foarte asemănătoare ca structură cu problema subşirului comun de lungime maximă este calculul distanţei Levensthein între două şiruri de caractere. Să presupunem că avem două şiruri de caractere, $x = x_1x_2 \dots x_m$ şi $y = y_1y_2 \dots y_n$. Operaţiile permise asupra unui şir de caractere sunt:

- ştergerea unui caracter oarecare din şir;
- inserarea unui caracter pe o poziţie oarecare din şir;
- înlocuirea unui caracter oarecare cu un altul.

Se cere să se determine numărul minim de operaţii prin care şirul x se poate transforma în şirul y. Acest număr se numeşte *distanţa Levensthein* dintre x şi y şi reprezintă o evaluare mult mai obiectivă a similarităţii a două şiruri decât distanţa Hamming (care pur şi simplu numără poziţiile pe care cele două şiruri diferă).

Exemplu: Pentru a transforma şirul *algoritm* în *aborigen* sunt necesare următoarele operaţii:
$algoritm[l \leftrightarrow b] \longrightarrow abgoritm[sterge\, g] \longrightarrow aboritm[t \leftrightarrow g] \longrightarrow$
$\longrightarrow aborigm[insereaza\, e] \longrightarrow aborigem[m \leftrightarrow n] \longrightarrow aborigen$

Este uşor de văzut că întotdeauna există o secvenţă de astfel de operaţii care să transforme un şir într-altul, deci nu se pune problema de fezabilitate. Structurarea în subprobleme este foarte asemănătoare cu cea de la problema subşirului comun de lungime maximă din secţiunea precedentă. Astfel, pentru oricare prefix $X_i = x_1 \dots x_i$ al lui x şi oricare prefix $Y_j = y_1 \dots y_j$ al lui y ne vom propune să determinăm secvenţa cea mai scurtă de operaţii pentru a-l transforma pe X_i în Y_j.

Să arătăm mai întâi că descompunerea în subprobleme dată de noi respectă principiul optimalităţii. Pentru aceasta, va trebui să găsim o modalitate de a construi soluţia optimă a unei probleme pe baza soluţiilor optime ale problemelor de dimensiune mai mică. Să considerăm două prefixe X_i şi Y_j ale şirurilor iniţiale, iar $s = s_1 s_2 \dots s_p$ secvenţa optimă de operaţii pentru a-l transforma pe primul în al doilea. Putem face acum următoarele observaţii:

- Dacă $x_i = y_j$, atunci $s = s_1 \dots s_p$ este o secvenţă optimală de transformări şi pentru X_{i-1}, Y_{j-1};
- Dacă $x_i \neq y_j$ şi ultima transformare din s este o operaţie de ştergere, atunci $s_1 \dots s_{p-1}$ este o secvenţă optimală de transformări pentru X_{i-1} şi Y_j;
- Dacă $x_i \neq y_j$ şi ultima transformare din s este o operaţie de inserare, atunci $s_1 \dots s_{p-1}$ este o secvenţă optimală de transformări pentru X_i şi Y_{j-1};
- Dacă $x_i \neq y_j$ şi ultima transformare din s este o operaţie de înlocuire, atunci $s_1 \dots s_{p-1}$ este o secvenţă optimală de transformări pentru X_{i-1} şi Y_{j-1}.

Cele patru observaţii de mai sus ne indică substructura optimă a problemei noastre. Determinarea secvenţei optimale de transformări pentru cele două şiruri X_i şi Y_j se reduce la determinarea secvenţei pentru una dintre perechile (X_{i-1}, Y_{j-1}), (X_{i-1}, Y_j) sau (X_i, Y_{j-1}).

Ca de obicei, vom calcula mai întâi care este numărul de operaţii pentru a-l tranforma pe x în y, urmând ca pe baza tabelului obţinut să determinăm efectiv care este secvenţa optimală de transformări. Să notăm cu d_{ij} distanţa Levensthein dintre X_i şi Y_j. Pentru $i = 0$, vom avea nevoie de j operaţii de inserare pentru a ajunge la Y_j (X_0 este şirul vid), deci $d_{0j} = j$; similar avem $d_{i0} = i$.

Să găsim acum relaţia de recurenţă pentru cazul general. Conform primei observaţii, dacă $x_i = y_j$, vom avea $d_{ij} = d_{i-1j-1}$. Dacă $x_i \neq y_j$ atunci d_{ij} va fi, funcţie de care dintre cele trei operaţii s-a aplicat, egal cu $d_{i-1j-1} + 1$, $d_{i-1j} + 1$, sau $d_{ij-1} + 1$. Vom alege desigur cea mai mică valoare. Astfel, formula de recurenţă pentru d_{ij} este:

$$d_{ij} = \begin{cases} j & pentru\, i = 0 \\ i & pentru\, j = 0 \\ d_{i-1j-1} & pentru\, x_i = y_j \\ 1 + \min\{d_{i-1j}, d_{ij-1}, d_{i-1j-1}\} & pentru\, x_i \neq y_j \end{cases}$$

Listing 14.18: Rezolvarea recurenţei pentru distanţa Levensthein

```
/**
 * Calculeaza formula de recurenta pentru matricea d.
 * @return distanta Levensthein dintre x si y
 * care se presupun a fi membri ai clasei care contine metoda.
 */
public static int distantaLevensthein()
{
  int m = x.length();
  int n = y.length();
  d = new int[m + 1][n + 1];

  for (int i = 0; i <= m; ++i)
  {
    d[i][0] = i;
  }

  for (int j = 0; j <= n; ++j)
  {
    d[0][j] = 0;
  }

  for (int i = 0; i < m; ++i)
  {
    for (int j = 0; j < n; ++j)
    {
      if (x.charAt(i) == y.charAt(j))
      {
        d[i + 1][j + 1] = d[i][j];
      }
      else
      {
        d[i+1][j+1] = 1 + min(d[i][j+1], d[i+1][j], d[i][j]);
      }
    }
  }

  return d[m][n];
}
```

Rezolvarea recurenţei de mai sus este foarte simplă, şi este realizată de metoda `distantaLevensthein()` din **Listing 14.18**.

Pentru a reconstitui secvenţa efectivă de operaţii prin care x se transformă în y vom folosi o parcurgere a matricei d asemănătoare cu cea de la problema subşirului comun de lungime maximă. Pornim de pe poziţia (m, n). Dacă $x_m = y_n$, atunci trecem pe poziţia $(m-1, n-1)$ fără nici o operaţie. Dacă

Figura 14.6: Elementele matricei d pentru şirurile *algoritm* şi *aborigen*. Elementele îngroşate reprezintă traseul corespunzător transformării optime dată la începutul paragrafului.

	λ	a	b	o	r	i	g	e	n
λ	**0**	1	2	3	4	5	6	7	8
a	1	**0**	1	2	3	4	5	6	7
l	2	1	**1**	2	3	4	5	6	7
g	3	2	**2**	2	3	4	4	5	6
o	4	3	3	**2**	3	4	5	6	7
r	5	4	4	3	**2**	3	4	5	6
i	6	5	5	4	3	**2**	3	4	5
t	7	6	6	5	4	3	**3**	**4**	5
m	8	7	7	6	5	4	4	4	**5**

$x_m \neq y_n$, atunci alegem poziţia învecinată de valoare minimă. Dacă valoarea minimă s-a obţinut pentru $(m-1, n-1)$, atunci ultima operaţie a fost de înlocuire; dacă valoarea minimă s-a obţinut pentru elementul de pe poziţia $(m-1, n)$, atunci ultima operaţie a fost de inserare, iar dacă valoarea minimă s-a obţinut pentru $(m, n-1)$, ultima operaţie a fost de ştergere.

Figura 14.6 prezintă matricea d pentru şirurile *algoritm* şi *aborigen*, iar elementele îngroşate corespund soluţiei date la începutul paragrafului. Algoritmul de reconstituire este asemănător cu cel de la problema subşirului comun de lungime maximă din paragraful precedent şi este redat în liniile 61-98 din **Listing 14.19**.

Listing 14.19: Soluţia completă a problemei distanţei Levensthein

```
1 import java.io.*;
2 import io.Reader;
3
4 public class Levensthein
5 {
6   private static String x;
7   private static String y;
8   private static int[][] d;
9
10  public static int distantaLevensthein()
11  {
12    int m = x.length();
13    int n = y.length();
14    d = new int[m + 1][n + 1];
15
```

```
    for (int i = 0; i <= m; i++)
    {
      d[i][0] = i;
    }

    for (int j = 0; j <= n; j++)
    {
      d[0][j] = j;
    }

    for (int i = 0; i < m; i++)
    {
      for (int j = 0; j < n; j++)
      {
        if (x.charAt(i) == y.charAt(j))
        {
          d[i+1][j+1] = d[i][j];
        }
        else
        {
          d[i+1][j+1] = 1 + min(d[i][j+1],
                      d[i+1][j],
                      d[i][j]);
        }
      }
    }

    System.out.println("Operatiile de transformare: ");
    determinaSecventa(m, n);

    return d[m][n];
  }

  private static int min(int a, int b, int c)
  {
    int m = Math.min(a, b);

    if (c < m)
    {
      m = c;
    }

    return m;
  }

  public static void determinaSecventa(int i, int j)
  {
    if (i != 0 && j != 0)
    {
      if (x.charAt(i-1) == y.charAt(j-1))
```

```
        {
          determinaSecventa(i-1, j-1);
        }
        else
        {
          int m = min(d[i-1][j-1],
                d[i-1][j],
                d[i][j-1]);

          if (m == d[i-1][j-1])
          {
            System.out.println("- inlocuire: " +
              x.charAt(i-1) + " cu " +
              y.charAt(j-1));
            determinaSecventa(i-1, j-1);
          }
          else if (m == d[i-1][j])
          {
            System.out.println("- stergere: " +
              x.charAt(i-1) +
              " de pe pozitia " + (i-1));
            determinaSecventa(i-1, j);
          }
          else
          {
            System.out.println("- inserare: " +
              y.charAt(j-1) +
              " pe pozitia " + i);
            determinaSecventa(i, j-1);
          }
        }
      }
    }

    public static void main(String args[])
    {
      System.out.print("x = ");
      x = Reader.readString();

      System.out.print("y = ");
      y = Reader.readString();
      System.out.println("Numarul minim de operatii: "
        + distantaLevensthein());
    }
}
```

Rezumat

Noţiuni fundamentale

programare dinamică: metodă de elaborare a algoritmilor care se aplică problemelor de optimizare în care soluţia optimă se construieşte pe baza soluţiei optime a subproblemelor.

problemă de optimizare: problemă în care se cere minimizarea sau maximizarea unei funcţii obiectiv (de exemplu, în cazul înmulţirii şirului de matrice, funcţia obiectiv este numărul de înmulţiri scalare necesare pentru a calcula produsul).

principiul optimalităţii: este numit adeseori şi *substructură optimă*. În esenţă, acest principiu afirmă că soluţia optimă a unei probleme de optimizare este construită din subsoluţii optime ale subproblemelor.

soluţie optimă: o soluţie pentru care se atinge valoarea optimă a funcţiei obiectiv în cazul unei probleme de optimizare. Soluţia optimă nu este neapărat unică.

subproblemă: problemă similară cu cea originală, dar de dimensiune mai mică. În cazul programării dinamice subproblemele se aleg astfel încât să respecte principiul optimalităţii.

Erori frecvente

1. Se confundă principiul optimalităţii cu reciproca lui, crezând că prin combinarea a două subsoluţii optime se obţine tot o soluţie optimă.

2. Se aplică metoda programării dinamice pentru subprobleme care nu respectă principiul optimalităţii.

3. Nu trebuie să ne avântăm să rezolvăm toate problemele care respectă principiul optimalităţii folosind programarea dinamică. Trebuie verificat înainte că nu putem aplica strategii mai simple, cum ar fi Greedy.

4. Suprapunerea apelurilor recursive trebuie evitată, deoarece există posibilitatea de a genera algoritmi exponenţiali.

5. Calculul complexităţii în timp a algoritmilor recursivi se face pe baza unei formule recurente. Nu vă bazaţi pe faptul că un apel recursiv are timp liniar.

Exerciţii

Pe scurt

1. Cărui tip de probleme se poate aplica metoda programării dinamice?

2. Care este enunţul principiului optimalităţii?

3. Este reciproca principiului optimalităţii adevărată?

4. Care este principala dificultate în rezolvarea problemelor de programare dinamică?

5. Construiţi matricea `l` şi evidenţiaţi drumul de lungime maximă pentru triunghiul:

4				
2	9			
11	2	1		
4	8	9	6	
7	10	1	12	4

6. Găsiţi un exemplu pentru care problema triunghiului din paragraful 14.3 admite mai multe soluţii optime. Construiţi matricea `l` atât pentru metoda înainte cât şi pentru metoda înapoi.

7. Găsiţi o parantezare optimă a produsului unui şir de matrice de dimensiuni $(4, 12, 5, 9, 2, 60, 8)$.

8. Determinaţi cel mai lung subşir comun pentru $(1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0)$ şi $(0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1)$.

9. Determinaţi distanţa Levensthein între şirurile $recursivitate$ şi $acuitate$.

Teorie

1. Demonstraţi că o parantezare corectă a unui şir de n matrice are exact $n - 1$ perechi de paranteze.

2. Care este modalitatea cea mai eficientă de determinare a parantezării optime a unui şir de matrice: enumerarea tuturor parantezărilor posibile ale produsului şi calculul numărului de operaţii pentru fiecare parantezare, sau rezolvarea recursivă a formulei de recurenţă 14.1? Justificaţi răspunsul.

Probleme

1. Toate problemele de programare dinamică prezentate în cadrul acestui capitol (problema triungiului, parantezarea unui şir de matrice, subşir comun de lungime maximă etc.) admit în unele situaţii mai multe soluţii optime. Totuşi, algoritmii prezentaţi de noi determină o *singură* soluţie optimă pentru fiecare problemă. Modificaţi metodele de determinare a soluţiei optime astfel încât acestea să afişeze *toate* soluţiile optime ale problemei.

   Indicaţie
   Toţi algoritmii de determinare a soluţiei optime se bazează pe parcurgerea ulterioară a matricei soluţie reconstruind traseul pe care soluţia optimă a fost obţinută. De exemplu, în cazul problemei triungiului, se porneşte din vârful matricei `l` şi se merge în jos sau dreapta-jos, funcţie de care element este mai mare. Totuşi, dacă cele două elemente sunt egale, înseamnă că soluţia optimă se poate construi mergând pe *ambele* variante. Rezolvarea corectă este aşadar să se parcurgă recursiv ambele variante şi să se afişeze soluţiile corespunzătoare fiecăreia. Este important de observat că situaţia în care elementele sunt egale poate să apară de mai multe ori în cadrul reconstruirii soluţiei. Practic, de câte ori apare această situaţie, atâtea soluţii optime admite problema.

2. **Problema bancomatului**.
   Se pune problema construirii unui bancomat care să fie capabil să furnizeze o anumită sumă clienţilor băncii. Pentru aceasta, aparatul dispune de un număr k de bancnote. Deşi fiecare dintre aceste bancnote există în cantitate nelimitată, se doreşte returnarea restului cu un număr minim de bancnote. Se cere ca dându-se o sumă S şi k numere reprezentând valorile bancnotelor $(v_1, v_2, \ldots v_n)$, să se determine numărul minim de bancnote cu care se poate plăti suma S, în cazul în care acest lucru este posibil. Să se determine şi bancnotele cu care suma este plătită.

   Observaţie
   Problema nu este trivială, întrucât bancnotele pot avea orice valoare naturală. De exemplu, nu se poate plăti suma 25 cu bancnote având valoarea 7 şi 13). De asemenea, strategia Greedy de a folosi monede de valoare maximă nu dă întotdeauna soluţia optimă (de exemplu, achitarea sumei 20 cu monede de valoare $6, 5, 4, 1$ ar genera soluţia $(6, 6, 6, 1, 1)$, care nu este optimă).

Indicaţie
Notăm cu b_i numărul minim de bancnote cu care poate fi plătită suma i. b_s ne va da chiar soluţia problemei (numărul minim de bancnote necesare pentru a plăti suma S). Prin convenţie, b_i va fi infinit dacă suma i nu poate fi platită cu bancnotele date. Se iniţializează elementele vectorului b cu valoarea infinit (`Integer.MAXINT` în cazul nostru). Pentru fiecare bancnotă cu valoarea v_k se pune $b_{v_k} = 1$. Apoi vom determina b_i, cu formula de recurenţă

$$b_i = \min_{i-v_k>0} \{b_{i-v_k}\} + 1 \quad i = 1, 2, \dots s$$

care exprimă faptul că suma i se poate obţine adăugând moneda v_k la monedele necesare pentru a obţine suma $i - v_k$. Reconstituirea soluţiei se face reţinând pentru fiecare i indicele k pentru care s-a obţinut valoarea minimă.

3. **Company party**.
Preşedintele unei companii doreşte să organizeze o petrecere cu angajaţii firmei. Aceştia se află într-o structură ierarhică de subordonare care este reprezentată sub forma unui arbore a cărui rădăcină este, desigur, chiar preşedintele. Pentru ca toată lumea să se simtă bine, nu trebuie să fie invitate două persoane care să se afle în relaţie directă de şef-subaltern. Mai mult decât atât, fiecare angajat are un atribut calculat de serviciul personal - *rata de sociabilitate*. Se doreşte ca suma ratelor de sociabilitate ale persoanelor invitate să fie maximă, asigurând astfel o petrecere mai mult decât reuşită. Ca o condiţie suplimentară, preşedintele trebuie să meargă la propria-i petrecere...

Indicaţie
Se consideră fiecare subarbore al ierarhiei date, calculând valoarea optimă în cazul în care vârful subarborelui este invitat şi în cazul în care nu este invitat. Dacă vârful arborelui este invitat la petrecere, atunci subalternii săi direcţi nu vor fi invitaţi. Dacă vârful nu este invitat, atunci nu există nici o restricţie pentru subalternii direcţi (care pot sau nu să fie invitaţi).

4. **Problema paragrafării.**
Se dau n cuvinte de lungime (număr de caractere) $l_1, l_2, \dots, l_n$. Trebuie să se aşeze aceste cuvinte în ordine într-un paragraf în care lungimea

unei linii este l. Fiecare linie din paragraf trebuie să înceapă cu un cuvânt. Distanţa optimă (numărul de spaţii) dintre două cuvinte este dată de un număr d. Distanţa minimă dintre două cuvinte este dată de numărul d_{min}. Toate cele n cuvinte au lungime mai mică decât l. Pentru fiecare linie se calculează un cost care se obţine prin însumarea abaterilor de la distanţa optimă dintre cuvinte la care se adună numărul de spaţii libere rămase după ultimul cuvânt până la finalul liniei. Costul ultimei linii este 0 chiar dacă aceasta este incompletă. Se defineşte costul paragrafării ca fiind suma costurilor liniilor care formează textul. Se cere să se furnizeze o aşezare a celor n cuvinte care să fie de cost minim.

Exemplu
Se dau 4 cuvinte de lungime respectiv 1,2,2,6, lungimea liniei fiind 8. Se consideră distanţa optimă $d = 2$, iar distanţa minimă $d_{min} = 1$. Paragrafarea optimă este (c reprezintă un caracter din cuvânt, iar s reprezintă un spaţiu):

$$cssccscc$$
$$ccccccss$$

având costul 1.

Indicaţie
Vom considera ca subproblemă de ordin i aranjarea în paragraf a cuvintelor $i, i+1, \dots, n$. Se observă uşor că dacă o soluţie optimă conţine o linie al cărei început este cuvântul i, atunci subsoluţia este optimă pentru subproblema de ordin i. Vom nota cu c_i costul paragrafării cuvintelor $i, i+1, \dots, n$. Evident, $c_n = 0$, deoarece costul ultimei linii este 0. Formula de recurenţă pentru şirul c este

$$c_i = \begin{cases} 0, \, pentru \, i = n \\ \min_{j=i,n-1} \{a_{ij} + c_{j+1}\}, \, altfel \end{cases}$$

unde a_{ij} este costul obţinut prin aşezarea cuvintelor $i, i+1, \dots j$ pe o linie (în situaţia în care cuvintele nu pot fi aşezate pe linie, costul se consideră a fi infinit). a_{ij} se află uşor calculând diferenţa dintre spaţiile disponibile $(l - \sum_{k=i}^{j} l_k)$ şi cele care ar trebui să fie pentru un cost 0 ($(j-i) \cdot d_{opt}$):

$$a_{ij} = \begin{cases} \infty, \, pentru \, \sum_{k=i}^{j} l_k + (j-i) \cdot d_{min} > l \\ \left| l - \sum_{k=i}^{j} l_k - (j-i) \cdot d_{opt} \right|, \, altfel \end{cases} .$$

Aflarea unei paragrafări optime se face reţinând cuvintele la care se trece pe linie nouă şi distribuind uniform spaţiile avute la dispoziţie în cadrul cuvintelor din aceeaşi linie.

5. **Mere, pere.**
Se consideră n camere distincte, situate succesiv una după cealaltă astfel încât din camera numărul i se poate trece doar în camera numărul $i + 1$ ($i = 1, 2, \ldots, n-1$). În fiecare cameră se află un anumit număr de mere şi de pere. O persoană având la dispoziţie un rucsac suficient de încăpător, iniţial gol, porneşte din camera 1, trece prin camerele $2, 3, \ldots, n$ şi iese. La intrarea în fiecare cameră persoana trebuie să descarce rucsacul şi să încarce fie toate merele, fie toate perele din camera respectivă, după care trece în următoarea cameră. Se presupune că pentru fiecare fruct transportat dintr-o cameră într-alta persoana consumă câte o calorie. Să se precizeze ce fructe trebuie să încarce persoana respectivă în fiecare cameră astfel încât după parcurgerea celor n camere să consume un număr minim de calorii şi să se precizeze acest număr.

Indicaţie
Strategia Greedy de a alege la fiecare pas cantitatea de fructe mai mică nu conduce întotdeauna la soluţia optimă (găsiţi un contraexemplu!). Principiul optimalităţii se formulează astfel: presupunem că avem o soluţie optimă pentru subproblema generată de camerele $i, i + 1, \ldots, n$; atunci subsoluţia acesteia pornind din camera $i+1$ este optimă pentru subproblema generată de camerele $i+1, i+2, \ldots, n$ **în care se pleacă cu fructele alese în camera $i + 1$ în cadrul soluţiei optime**. Cu alte cuvinte, dacă în camera $i + 1$ s-au ales mere, atunci subsoluţia respectivă este optimă pentru subproblema generată de camerele $i+1, i+2, \ldots, n$ în care se presupune că se porneşte cu mere (dacă se porneşte cu pere se poate obţine o soluţie mai bună).

Vom nota cu m_i numărul merelor şi cu p_i numărul perelor din camera $i, i = 1, 2, \ldots, n$. De asemenea, vom nota cu cm_i costul optim (numărul minim de calorii) obţinut când se pleacă cu mere din camera i şi cu cp_i costul optim obţinut când se pleacă cu pere din camera i. Evident, soluţia problemei este dată de $\min(cp_1, cm_1)$. Formula de recurenţă este dată de observaţia că dacă plecăm cu un anumit tip de fruct din camera i, putem

să schimbăm acel tip de fruct în oricare dintre camerele $i+1, i+2, \ldots, n$:

$$cm_i = \begin{cases} m_n, \ i = n \\ \min \begin{cases} m_i + cp_{i+1} \\ m_i + (m_i + m_{i+1}) + cp_{i+2} \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ m_i + (m_i + m_{i+1}) + \ldots + (m_i + \ldots m_{n-1}) + cp_n \\ m_i + (m_i + m_{i+1}) + \ldots + (m_i + \ldots m_n) \end{cases} \end{cases}$$

Analog se obţine formula de recurenţă şi pentru pere.

6. **Mere¸ pere - problema generalizată.**
   Generalizaţi problema anterioară pentru cazul în care în fiecare cameră se găsesc t tipuri de fructe.

7. **Compararea secvenţelor de ADN.**
   Se numeşte secvenţă de ADN un şir format din caracterele A,C,T şi G (corespunzătoare elementelor fundamentale ale codului genetic: adenină, citozină, timină şi guanină). O problemă de o deosebită importanţă în genetică este compararea a două secvenţe de ADN (de exemplu, pentru stabilirea paternităţii). Compararea a două secvenţe de ADN nu se face însă direct, existând posibilitatea alinierii celor două şiruri prin inserarea de caractere '-' (după aliniere cele două şiruri au aceeaşi dimensiune). De exemplu, dacă avem secvenţele TGCGAT şi TAGCAG, o posibilă alinere este:

$$T - GCGAT$$

$$TAGC - AG$$

   Se acordă 3 puncte dacă două caractere care se află unul sub celălalt sunt egale şi se acordă o penalizare de un punct dacă cele două caractere sunt diferite. Pentru exemplul de mai sus se acordă 4*3-3*1=9 puncte.

   Fiind date două şiruri ADN, A şi B (nefiind obligatoriu ca acestea să aibă aceeaşi dimensiune), se cere să se determine o aliniere pentru care punctajul acordat este maxim. (Pentru exemplul precedent, alinierea dată este optimă).

Indicaţie
Această problemă seamănă întrucâtva cu problema determinării subşirului comun de lungime maximă (paragraful 14.6). O primă idee de rezolvare ar fi să determinăm subşirul comun de lungime maximă pentru cele două şiruri şi apoi să potrivim elementele comune din cele două şiruri unele sub altele, folosindu-ne de '-' atât sus cât şi jos. Această soluţie ar fi corectă dacă nu ar exista penalizări pentru nepotriviri. Cu toate acestea, structura de optimalitate este asemănătoare cu cea a problemei subşirului comun de lungime maximă, diferind doar relaţia de recurenţă. Şi aici vom considera prefixe ale celor două şiruri, iar pentru fiecare pereche de prefixe vom determina alinierea optimală. Prin aliniere se înţelege o reprezentare a ieşirii cerute de problemă în care se inserează caractere '-' pentru ca şirurile să aibă aceeaşi lungime.

Fie s_{kl} scorul maxim ce se poate obţine prin alinierea prefixelor $A_1 \ldots A_k$ şi $B_1 \ldots B_l$. Dacă $A_k = B_l$, atunci s_{kl} va fi $s_{k-1l-1} + 3$, deoarece se acordă 3 puncte pentru două caractere identice aşezate unul sub altul. Dacă cele două caractere nu sunt egale, atunci putem obţine s_{kl} fie din s_{k-1l}, fie din s_{kl-1} sau din s_{k-1l-1} corespunzător respectiv situaţiei în care adăugăm un '-' la şirul A, un '-' la şirul B sau câte un caracter diferit de '-' la ambele şiruri. Pentru fiecare dintre aceste trei cazuri va trebui acordată o penalizare pentru nepotrivirea caracterelor. Desigur, vom alege varianta al cărei cost este maxim.

Formula de recurenţă pentru matricea s

$$s_{kl} = \begin{cases} -k, \; pentru \, l = 0 \\ -l, \; pentru \, k = 0 \\ s_{k-1l-1} + 3, \; pentru \, A_k = B_l \\ \max\{s_{k-1l}, s_{kl-1}, s_{k-1l-1}\} - 1, \; pentru \, A_k \neq B_l \end{cases}$$

Reconstituirea soluţiei se face uşor parcurgând matricea s de la s_{mn} până la s_{00}. Dacă s_{kl} provine din s_{k-1l-1}, atunci este vorba de o aşezare unul sub altul a două caractere ADN (deci nu '-'), momentan nefiind important dacă A_k este egal cu B_l sau nu. Dacă s_{kl} provine din s_{k-1l}, atunci trebuie inserat un '-' în şirul B, altfel el trebuie inserat în A.

8. **Problema vrăjitorului.**
Indiana Jones intră într-un labirint unde găseşte un vrăjitor. Acesta îi pune în faţă n lăzi, fiecare ladă conţinând un anumit număr precizat (m_i)

de monede de aur. Vrăjitorul îi cere să aleagă nişte lăzi astfel încât suma monedelor din acestea să fie divizibilă cu n, altfel nu va putea pleca cu ele. Evident, Indiana Jones doreşte să ia un număr cât mai mare de monede de aur. Elaboraţi un algoritm care să-l ajute pe Indiana Jones să plece cu cantitatea maximă de monede, respectând cerinţele vrăjitorului.

Indicaţie

Trebuie să arătăm în primul rând că problema noastră admite soluţie, cu alte cuvinte există cel puţin o combinaţie de lăzi pentru care suma monezilor este divizibilă cu n. Notăm cu s_i suma monezilor din lăzile $1, 2, \ldots i$

$$s_i = \sum_{j=1}^{i} m_i$$

şi cu r_i restul împărţirii lui s_i la n. Evident, r_i va lua valori întregi între 0 şi $n-1$. Dacă există un i pentru care avem $r_i = 0$, atunci rezultă că s_i se divide cu n, deci lăzile $1, 2, \ldots, i$ reprezintă o soluţie a problemei. Dacă nu există nici un i pentru care $r_i = 0$, atunci rezultă că r_i ia valori între 1 şi $n-1$, deci vor exista cel puţin două componente r_i şi r_j din şirul r (care are n componente) care să fie egale. Aceasta înseamnă că $s_j - s_i$ se divide cu n, deci lăzile $i+1, \ldots, j$ reprezintă o soluţie a problemei. Am arătat deci că problema are întotdeauna măcar o soluţie. Desigur, această metodă de alegere a lăzilor nu garantează optimalitatea soluţiei, motiv pentru care vom folosi metoda programării dinamice pentru a rezolva problema.

Vom nota cu M_{ij} suma maximă de monezi care se poate obţine din primele i lăzi şi care să dea restul j la împărţirea cu n. Prin convenţie, M_{ij} va fi infinit dacă nu se poate obţine o sumă care să dea restul j la împărţirea cu n. Scopul nostru este de a-l calcula pe M_{n0}. Este uşor de observat că M_{1j} este egal cu m_1 dacă m_1 se divide cu j, şi infinit în caz contrar. Totuşi, în cazul în care m_1 nu se divide cu n, vom considera M_{10} ca fiind 0. În cazul general, pentru a calcula M_{kl} ne vom folosi de informaţia determinată pentru primele $k-1$ lăzi. Lada numărul k poate sau nu să apară în soluţia subproblemei M_{kl}. Dacă nu este luată, atunci vom avea $M_{kl} = M_{k-1l}$. Dacă lada numărul k este luată, atunci M_{kl} va fi egal cu

$$M_{jp} + m_k$$

unde j este mai mic decât k, iar $p+m_k$ trebuie să dea restul l la împărţirea la n. Formula de mai sus rezultă din observaţia că M_{jp} reprezintă suma

maximă care se poate obţine din primele j lăzi care să fie divizibilă cu p. Dacă acestei sume i se adaugă m_k, se obţine o sumă care prin împărţirea la n va da restul l, adică chiar M_{kl} (la acest pas se aplică principiul optimalităţii). Dintre toate aceste variante o vom alege, desigur, pe cea mai convenabilă

$$M_{kl} = \max\{M_{k-1l}, \max_{j=1}^{k-1}\{M_{jp} + m_k | j < k \, si \, l = p + m_k \, (mod \, n)\}\}$$

Valoarea lui p din formula de mai sus poate fi calculată cu uşurinţă după formula $p = (l - m_k) \, mod \, n$ (atenţie la cazul în care $l - m_k$ este negativ - puteţi adăuga pentru siguranţă un n la sumă). Reconstituirea soluţiei se face reţinând pentru fiecare element M_{kl} o valoare p_{kl} care indică lada j pentru care s-a obţinut valoarea optimă, sau este 0 dacă $M_{kl} = M_{k-1l}$. cu

15. Metoda Branch & bound

> Trebuie să fii foarte atent dacă nu ştii unde mergi, pentru că rişti să nu ajungi acolo.
>
> Yogi Berra

Pe parcursul acestui capitol vom vedea:

- În ce constă metoda Branch & bound de elaborare a algoritmilor;
- Care este legătura dintre Branch & bound şi Backtracking;
- Abordarea unei probleme clasice de Branch & bound, cunoscută sub numele de jocul de puzzle cu 15 elemente.

15.1 Prezentare generală

Metoda Branch & bound este asemănătoare metodei Backtracking, în sensul că încearcă să facă o căutare exhaustivă inteligentă în spaţiul soluţiilor (numit în acest context şi *spaţiul stărilor*) problemei. Comparativ cu Branch & bound, Backtracking acţionează "orbeşte", în sensul că la fiecare pas se alege (la întâmplare) o componentă în vectorul soluţie care respectă condiţiile de continuare. Singura condiţie la Backtracking este aşadar respectarea condiţiilor de continuare: toate posibilele configuraţii care respectă această condiţie sunt privite în mod egal şi se alege arbitrar una dintre ele. Alegerea în cazul Branch & bound este ceva mai rafinată: fiecărei configuraţii viabile i se acordă o pondere care reprezintă o estimare a valorii acelei configuraţii. La fiecare pas vom alege configuraţia a cărei estimare este optimă. Ca şi în cazul metodei Backtracking, problemele rezolvabile prin această metodă generează un spaţiu al stărilor,

în care se găsesc toate configuraţiile viabile posibile. Spaţiul stărilor poate fi reprezentat în acest caz sub forma unui arbore, în care rădăcina arborelui este configuraţia iniţială a problemei, în timp ce configuraţiile finale constituie, în cazul în care există, frunzeale arborelui. Este posibil ca pentru o anumită problemă să nu existe o soluţie, adică să nu existe o secvenţă de mutări posibile, prin care să se ajungă de la o configuraţie iniţială la o configuraţie dorită (finală). Asemănarea cu Backtracking este şi acum evidentă. În cazul metodei Backtracking spaţiul stărilor era reprezentat sub forma unei *liste* care începea cu configuraţia iniţială şi se încheia cu configuraţia finală în care nu se mai putea aplica nici una dintre cele patru transformări (atribuie şi avansează, încercare eşuată, revenire, revenire după găsirea unei soluţii). Diferenţa esenţială dintre Backtracking şi Branch&bound apare în cadrul pasului *atribuie şi avansează*. În cazul Backtracking alegeam o configuraţie arbitrară care respecta condiţiile de continuare. În cazul Branch & bound considerăm *toate* configuraţiile pe care putem avansa şi o alegem pe cea care pare să fie mai promiţătoare.

Branch & bound găseşte soluţia optimală prin alegerea celei mai bune soluţii de moment. Dacă soluţia parţială curentă nu este mai potrivită decât cea mai bună soluţie parţială disponibilă la un moment dat, atunci ea este abandonată. De exemplu, să presupunem că se doreşte aflarea celui mai scurt drum de la Braşov la Constanţa şi că cea mai scurtă cale descoperită până la un moment dat este de 400 de kilometri. Apoi, să presupunem că vrem să considerăm posibilele rute care trec prin Galaţi. Dacă cel mai scurt drum din Braşov la Galaţi are 380 de kilometri, iar distanţa dintre Galaţi şi Constanţa este de 50 de kilometri, atunci nu are nici un sens să căutăm drumuri către Constanţa care trec prin Galaţi, pentru că ele vor fi tot timpul mai lungi decât cea mai scurtă cale cunoscută până în acel moment ($380 + 50 > 400$). De aceea, rutele din Braşov către Constanţa prin Galaţi nu vor mai fi explorate. Cu alte cuvinte, se încearcă parcurgerea subarborilor din spaţiul soluţiilor care pot conduce la un rezultat favorabil, evitându-se subarborii despre care se află că nu conduc la rezultate optime.

15.1.1 Fundamente teoretice

Metoda Branch & bound utilizează câteva noţiuni a căror înţelegere este strict necesară pentru a putea deprinde mecanismul de funcţionare a metodei.

Unei probleme de tipul Branch & bound i se asociază un *arbore oarecare* (nu neapărat arbore binar), în care fiecare nod reprezintă o *configuraţie*. Configuraţia iniţială reprezintă rădăcina arborelui şi conţine datele de intrare ale problemei. Prin efectuarea operaţiilor (mutărilor) permise asupra configuraţiei iniţiale se obţin alte configuraţii, care vor constitui nodurile aflate pe nivelul 2 al

arborelui. Procedeul se repetă, obţinându-se noi şi noi configuraţii, care conduc la crearea unei structuri arborescente, care va conţine soluţia problemei (dacă aceasta există).

Succesorul unui nod reprezintă o configuraţie la care se ajunge prin aplicarea nodului respectiv a uneia dintre mutările permise de problemă. În cele mai multe probleme, un nod are mai mulţi succesori.

Expandarea unui nod constituie acţiunea de determinare a tuturor succesorilor nodului respectiv. Nodul care este expandat este un nod de tip "tată", în timp ce nodurile rezultate în urma expandării sunt noduri de tip "fiu".

Nodul răspuns reprezintă configuraţia la care se doreşte să se ajungă (altfel spus, configuraţia finală). Acesta este obţinut prin aplicarea unei succesiuni de mutări configuraţiei iniţiale. Există şi situaţii în care nu există o succesiune de mutări prin care să se ajungă la configuraţia finală. În acest caz, spunem că problema nu are soluţie.

Nodul terminal (*frunză*) este un nod care nu are succesori. Dacă la un moment dat nu se mai poate realiza nici o mutare pe configuraţia curentă sau s-a ajuns la configuraţia finală, atunci ea nu va mai avea nici un succesor.

Există două tipuri de noduri în arborele asociat problemei Branch & bound: noduri *active* şi noduri *expandate*.

Nodurile *active* reprezintă mulţimea de noduri obţinută prin expandarea unui nod. Aceste noduri sunt denumite şi noduri *încă neexpandate*.

Un nod *expandat* este nodul pe care se realizează operaţia de expandare. Practic, pe configuraţia reprezentată de acest nod se aplică toate operaţiile permise de problemă, obţinându-se o mulţime de noduri active. Alegerea nodului expandat este o problemă dificilă, deşi nodul expandat este ales doar dintre elementele mulţimii nodurilor active existente în acel moment. Primul nod expandat este reprezentat de configuraţia iniţială. Următoarele noduri expandate sunt alese pe baza unui procedeu care va fi prezentat mai târziu. Nodul expandat este întâlnit în literatura de specialitate şi sub numele de nod *E* (engl. *E-node*) sau nod *curent*.

Nodurile active sunt stocate într-o listă. Modul de reprezentare al listei nodurilor active ne oferă informaţii despre modul de parcurgere al arborelui soluţiilor. Dacă nodurile active sunt păstrate într-o stivă (structură LIFO, Last In First Out), atunci arborele spaţiului de stări va fi parcurs în adâncime (engl. DFS = Depth First Search), iar dacă nodurile sunt păstrate într-o coadă (structură FIFO, First In First Out), atunci arborele spaţiului de stări va fi parcurs în lăţime (engl. BFS = Breadth First Search).

Nu există un criteriu, care odată aplicat, să permită găsirea următorului nod care trebuie expandat. Din acest motiv se apelează la un procedeu specific inteligenţei artificiale, prin care se asociază fiecărui nod o funcţie *c* (*funcţie*

de cost). Ca şi funcţia de continuare din cazul metodei Backtracking, aceasta este o funcţie de limitare, utilizată pentru evitarea generării unor subarbori care nu au cum să conducă la nodul răspuns. De aici şi numele metodei: *branch* înseamnă ramură, despărţitură, în timp ce *bound* înseamnă margine, limită, graniţă. Alegerea acestei funcţii este partea cea mai dificilă a metodei, pe ea bazându-se şi selectarea următorului nod expandat.

Pe scurt, un algoritm Branch & bound funcţionează la modul următor: din lista nodurilor active se va alege unul care devine nod expandat. Acest nod este următorul nod care va fi prelucrat. Nu pot exista mai multe noduri expandate în acelaşi timp. În momentul în care un nod devine nod expandat, se vor genera (determina) toate nodurile descendente (noduri succesor), acestea devenind noduri active. Ele se vor adăuga listei nodurilor active existente. Odată generaţi descendenţii nodului expandat, va fi ales acela care este situat cel mai aproape de configuraţia finală sau, mai corect spus, cel care are cele mai mari şanse de a se afla mai aproape de starea finală. Pentru a afla care este acest nod, se defineşte funcţia de cost c pe mulţimea nodurilor din arborele spaţiului soluţiilor. Această funcţie diferă de la problemă la problemă şi trebuie aleasă în funcţie de cerinţele problemei respective, dar descoperirea ei este un pas important spre rezolvarea problemei. Odată ce am determinat funcţia de cost, vom alege dintre nodurile active pe cel cu costul minim (pentru care funcţia ia valoarea minimă), care devine astfel nod expandat. Parcurgerea de acest tip poartă numele de *căutare LC* (căutare Least Cost = căutare după costul minim). Altfel spus, atunci când căutăm o soluţie în arborele generat de spaţiul configuraţiilor folosind funcţia de limitare, spunem că efectuăm o căutare LC a soluţiei problemei, indiferent dacă arborele în sine este parcurs în adâncime (DFS) sau în lăţime (BFS). Căutarea LC nu *înlocuieşte* căutările în lăţime sau în adâncime, ci doar le *îmbunătăţeşte*. Funcţia de cost nu poate fi determinată în mod precis, deoarece determinarea ei exactă este echivalentă cu rezolvarea problemei iniţiale. Chiar dacă nu putem determina o funcţie de cost infailibilă, există câteva sugestii care pot fi urmate în definirea ei:

- să poată fi calculată pentru un nod doar pe baza informaţiilor deţinute de nodurile intermediare aflate pe drumul către rădăcina arborelui;

- să fie independentă de generarea nodurilor active ale nodului expandat.

Figura 15.1: Un posibil aranjament al jocului de puzzle cu 15 elemente.

1	2	3	4
5	6		8
9	10	7	11
13	14	15	12

Figura 15.2: Aranjamentul final al jocului de puzzle cu 15 elemente.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

15.2 Un exemplu: Puzzle cu 15 elemente

15.2.1 Enunţul problemei

Puzzle cu 15 elemente este un joc distractiv inventat de Sam Loyd în 1878, care constă într-un tabel cu 16 celule (căsuţe), dintre care 15 celule sunt numerotate de la 1 la 15, iar o celulă rămâne liberă (**Figura** 15.1).

Regulile acestui joc sunt simple. Iniţial se dă un aranjament de tipul celui din **Figura** 15.1, iar scopul jocului este de a transforma acest aranjament prin transformări (mutări) succesive în aranjamentul din **Figura** 15.2.

Aranjamentul iniţial se mai numeşte *configuraţie iniţială* sau *stare iniţială*, iar cel final *configuraţie finală* sau *stare finală*. Pentru a ajunge de la configuraţia iniţială la cea finală este permis doar un singur tip de mutare: o celulă vecină cu celula liberă poate să facă schimb de poziţii cu aceasta. Astfel, să considerăm aranjamentul din **Figura** 15.1 ca fiind configuraţia iniţială. Atunci există patru mutări permise de regulamentul jocului: se poate muta celula liberă în locul uneia din celulele care conţin valorile 3, 6, 8, 7, ca în **Figura** 15.3

După cum se poate observa, se consideră a fi celule vecine ale celulei libere, doar cele care au o latură comună cu celulă respectivă (în cazul configuraţiei noastre iniţiale este vorba de căsuţele cu valorile 3, 6, 7, 8). Căsuţele cu valorile 2, 4, 10, 12 *nu* sunt considerate celule vecine ale celulei libere. Evident, nu tot timpul sunt posibile toate cele patru mutări. În situaţia în care celula liberă nu

Figura 15.3: Mutările permise pentru celula liberă din tabela 15.1.

sus

1	2		4
5	6	3	8
9	10	7	11
13	14	15	12

dreapta

1	2	3	4
5	6	8	
9	10	7	11
13	14	15	12

jos

1	2	3	4
5	6	7	8
9	10		11
13	14	15	12

stânga

1	2	3	4
5		6	8
9	10	7	11
13	14	15	12

are patru celule vecine (de exemplu, dacă se află pe prima linie, sau pe prima coloană) mutările respective nu sunt posibile.

Asemănător, pe fiecare dintre cele patru configuraţii obţinute se pot realiza alte mutări, generându-se astfel un nou set de configuraţii. Toate aceste configuraţii poartă numele de stări ale jocului. O stare poate fi obţinută pornind de la starea iniţială căreia i se aplică o secvenţă de mutări permise. Spaţiul stărilor unei configuraţii iniţiale reprezintă mulţimea tuturor stărilor care pot fi obţinute pornind de la starea iniţială.

Generarea configuraţiilor continuă până când se ajunge la configuraţia finală. Uneori este posibil ca starea finală să nu poată fi obţinută pornind de la starea iniţială, caz în care problema noastră nu va avea soluţie.

15.2.2 Rezolvarea problemei

Cea mai directă cale de a rezolva un astfel de puzzle este ca pornind de la starea iniţială să se genereze toate configuraţiile intermediare până când se obţine configuraţia finală, sau s-a epuizat întreg spaţiul de căutare. Este uşor de observat că numărul total al configuraţiilor jocului este de $16! \approx 20.9 * 10^{12}$. Prin urmare, spaţiul stărilor pentru problema noastră este foarte mare şi o căutare exhaustivă este sortită eşecului.

Înainte de a încerca să căutăm configuraţia finală în cadrul spaţiului stărilor, este util să determinăm dacă starea finală *poate* fi obţinută printr-o secvenţă de mutări din configuraţia iniţială. Pentru aceasta vom nota căsuţele de la 1 la 16. Astfel, în configuraţia iniţială din **Figura** 15.1 celula 1 conţine valoarea 1, celula 11 conţine valoarea 7, celula 12 conţine valoarea 11, etc. Pentru configuraţia

Figura 15.4: Valoarea variabilei x, funcţie de poziţia celulei libere în configuraţia iniţială

	*		*
*		*	
	*		*
*		*	

finală, cele două valori sunt egale: celula 1 conţine valoarea 1, celula 2 conţine valoarea 2 etc... Celulei libere i se atribuie valoarea 16.

Considerăm funcţia $position(i)$ ca fiind numărul celulei care conţine valoarea i. De exemplu, în configuraţia iniţială prezentată în **Figura** 15.1, avem:

- $position(3) = 3$
- $position(7) = 11$
- $position(11) = 12$ etc.

Pentru celula liberă avem $position(16) = 7$, ceea ce înseamnă că spaţiul liber se află pe poziţia 7 în configuraţia iniţială.

De asemenea, considerăm funcţia $less(i)$ ca fiind numărul de celule j, cu $j < i$, pentru care $position(j) > position(i)$. Cu alte cuvinte, această funcţie calculează câte celule cu valori mai mici decât valoarea celulei curente se găsesc după ea în configuraţia iniţială. De exemplu, în **Figura** 15.1, pentru celula care conţine valoarea 8, există o celulă cu valoare mai mică aflată pe o poziţie mai mare, şi anume, celula cu valorea 7. Aşadar, *less(8)*=1. Analog, se pot determina şi celelalte valori: *less(13)=1*, *less(5)=0, less(11)*=0, etc.

Ultima consideraţie se referă la poziţia spaţiului liber în cadrul configuraţiei iniţiale. Considerăm o variabilă *x* care are valoarea 1, dacă spaţiul liber se află în configuraţia iniţială pe una din poziţiile marcate cu * în **Figura** 15.4, sau 0, dacă spaţiul liber se află pe una dintre poziţiile nemarcate în aceeaşi figură.

Având aceste notaţii, putem enunţa următorul rezultat:

Teorema 15.2.1 *Configuraţia finală poate fi obţinută din configuraţia iniţială dacă şi numai dacă*

$x + \sum_1^{16} less(i)$ *este un număr par.*

Demonstraţia acestei teoreme se bazează pe observaţii simple asupra configuraţiilor posibile şi o puteţi găsi pe multe situri care tratează această problemă

(de exemplu `http://kevingong.com/Math/SixteenPuzzle.html`). Putem folosi această teoremă pentru a determina dacă starea finală se află în spaţiul stărilor asociat stării iniţiale. Dacă starea finală poate fi obţinută plecând de la starea iniţială, atunci are sens să începem căutarea secvenţei de mutări permise care ne conduce la starea finală.

Pentru a realiza această căutare este util să organizăm spaţiul stărilor într-un arbore. Rădăcina arborelui este starea (configuraţia) iniţială. Copiii fiecărui nod X al arborelui reprezintă stări care pot fi obţinute din starea X printr-o mutare permisă de regulamentul jocului. Pentru simplitatea notaţiei, este mai convenabil să ne gândim la o mutare ca fiind mai degrabă o mutare a celulei libere decât a unei alte celule. Prin urmare, celula liberă poate fi mutată la fiecare mutare, fie în sus, în jos, la stânga sau la dreapta, dacă este posibil.

Iată cum ar arăta arborele spaţiului de stări după câteva mutări în configuraţia iniţială:

După cum se poate observa şi din **Figura** 15.5, prin mutări succesive s-au obţinut diverse stări intermediare, care au creat o structură arborescentă: starea A a generat stările B, C, D şi E prin mutări în sus, respectiv la dreapta, în jos şi la stânga. Cele patru stări nou create, la rândul lor, au generat alte stări (de exemplu, starea D a generat stările J şi K prin mutări la dreapta şi în jos). În final din starea J s-a generat starea M prin mutarea spaţiului liber în jos, atingându-se starea finală, situaţie în care generarea unor noi mutări este stopată. Se poate observa că au fost necesare trei mutări pentru a atinge starea finală pornind de la starea iniţială. Figura anterioară nu este completă, în sensul că, pentru a păstra simplitatea, am ales să nu prezentăm toate mutările care ar fi fost posibile. De exemplu, configuraţia D generează de fapt patru alte stări (prin mutări în sus, în jos, la stânga şi la dreapta ale spaţiului liber), dintre care noi am prezentat doar două (J şi K).

Dacă am fi generat arborele stărilor folosind una dintre cele două posibilităţi, parcurgerea în lăţime (în care se generează toate stările obţinute din stările B, C, D, E, apoi toate stările pentru F, G, H etc.) sau cea în adâncime (în care se generează toate stările obţinute din starea B, apoi toate stările obţinute din starea F etc.), ne-am fi îndepărtat simţitor de soluţia problemei. Acest lucru este pus în evidenţă foarte clare de **Figura 15.5**. Asta se întâmplă pentru că, folosind parcurgerea spaţiului de configuraţii în lăţime sau în adâncime, căutarea soluţiei este "oarbă", adică nu ţine cont de configuraţia iniţială şi nici de cele intermediare obţinute, pentru a vedea care dintre ele este mai aproape de configuraţia finală. Practic, procedeul ar genera aceeaşi secvenţă de stări indiferent de starea iniţială a problemei. Căutarea în adâncime corespunde în mod precis parcurgii spaţiului de căutare rezultat în urma aplicării metodei *Backtracking*. Coborârea în arbore se asociază pasului *atribuie şi avansează*, iar revenirea la nodul părinte

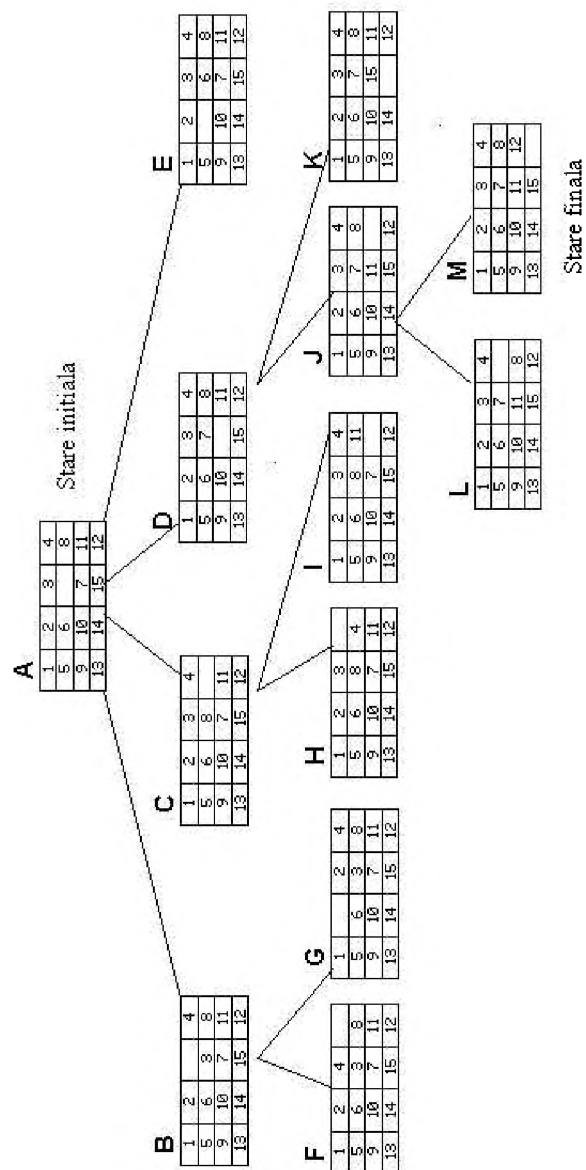


Figura 15.5: O parte a arborelui stărilor asociat stării iniţiale din **Figura** 15.1. Se pot observa starea iniţială şi cea finală.

se asociază pasului de *revenire*.

Pentru a evita căutarea oarbă a soluţiei, am dori să utilizăm o metodă de căutare "inteligentă". Această metodă inteligentă ar trebui să caute nodul răspuns (configuraţia finală) şi să adapteze calea către acesta, în funcţie de starea iniţială de la care pornim căutarea. O metodă pentru evitarea căutării brute este să asociem o funcţie de cost fiecărui nod din spaţiul stărilor, funcţie care să scoată cumva în evidenţă nodurile care sunt mai "promiţătoare", deci care au şanse mai mari să ne conducă la soluţia problemei.

Aşadar, fiecărui nod X din arborele stărilor îi asociem o funcţie c, iar $c(X)$ reprezintă costul nodului X. Funcţia c o definim ca o sumă de alte două funcţii f şi g, unde $f(X)$ o definim ca fiind lungimea drumului de la rădacina arborelui de stări (altfel spus, de la configuraţia iniţială) până la nodul X, iar $g(X)$ este o estimare a lungimii căii celei mai scurte de la nodul X la un nod răspuns aflat în subarborele de rădăcină X. O posibilă alegere pentru $g(X)$ este numărul de celule nelibere care nu se află pe poziţia din configuraţia finală. Aceasta este o alegere naturală, din moment ce este clar că pentru a ajunge la configuraţia finală trebuie realizate cel puţin $g(X)$ mutări pe starea X. Pentru o mai bună înţelegere a funcţiei de cost, iată câteva valori calculate pentru arborele de stări din **Figura** 15.5: $c(B) = 1 + 4 = 5$ ($f(B) = 1$ pentru că nodul B se află pe nivelul 2 în arbore, deci la o distanţă de un (1) nivel faţă de rădăcină, iar $g(B) = 4$ pentru că sunt 4 valori care nu se află în poziţiile lor finale: $3, 7, 11, 12$), $c(C) = 1 + 5 = 6$, $c(D) = 1 + 2 = 3$ etc.

Folosind funcţia c definită anterior, vom realiza o căutare LC a nodului răspuns pe baza algoritmului descris în paragraful 15.1.1. După cum s-a putut observa în cazul căutărilor nodului răspuns prin intermediul parcurgerii standard în lăţime sau în adâncime a arborelui de stări, selecţia nodului expandat nu s-a realizat ţinând cont de şansele de a ajunge la nodul răspuns într-un timp cât mai redus. Căutarea nodului răspuns este "îmbunătăţită" din punct de vedere al vitezei prin utilizarea unei funcţii inteligente de clasificare a nodurilor, aşa cum este cazul funcţiei de cost c. Următorul nod expandat va fi selectat, în algoritmul pe care îl propunem, nu la întâmplare (a se citi "indiferent de configuraţia reprezentată de nodul respectiv"), ci pe baza valorilor pe care posibilele noduri expandabile le vor avea prin aplicarea funcţiei de cost.

Căutarea LC porneşte de la configuraţia iniţială (**Figura** 15.1). Conform algoritmului, acest nod este primul nod expandat (nodul A din **Figura** 15.5). Se generează toţi fiii nodului expandat, adică nodurile B, C, D şi E. Aceste patru noduri reprezintă noduri active şi vor fi adăugate în lista nodurilor active. Din lista nodurilor active existente, se va alege viitorul nod expandat. Pentru aceasta se calculează $c(B)$, $c(C)$, $c(D)$, $c(E)$, alegându-se nodul pentru care funcţia c are valoarea minimă. Cum $c(B) = 5$, $c(C) = 6$, $c(D) = 3$, $c(E) = 1 + 4 = 5$,

rezultă că următorul nod expandat este nodul D. Nodul proaspăt expandat este eliminat din lista nodurilor active. În continuare, pentru acest nod se generează toţi fiii (nodurile J şi K din **Figura** 15.5 sunt doar o parte dintre fiii nodului expandat, la ceilalţi renunţându-se din lipsă de spaţiu), care devin noduri active şi sunt adăugaţi în lista nodurilor active (aceasta va conţine nodurile B, C, E, J şi K). Pentru nodurile active existente se calculează din nou valorile funcţiei c, obţinându-se valoarea minimă pentru nodul J, care devine noul nod expandat, fiind eliminat din lista nodurilor active. Procedeul se aplică într-un mod similar, până când se obţine nodul răspuns, dacă problema are soluţie. În final, în cazul în care problema are soluţie, se obţine secvenţa de mutări care a dus la obţinerea nodului răspuns. În cazul nostru, această secvenţă este A, D, J, M.

Lipsa spaţiului a făcut ca arborele de stări asociat configuraţiei iniţiale să nu conţină toate mutările posibile.

În final, prezentăm implementarea Java a problemei puzzle-ului cu 15 elemente:

```
import java.util.*;
import io.Reader;

/**
 * Rezolvarea jocului de Puzzle cu 15 elemente.
 */
public class Puzzle
{
  //pastreaza valorile functiei f ptr. fiecare nod activ
  public static Vector valoriF = new Vector();

  //distanta de la radacina la un nod
  public static int distanta = 0;

  public static void main(String[] args)
  {
    System.out.println("Introduceti configuratia initiala " +
      "(ptr. spatiul gol tastati valoarea 16): ");

    int n = 4;
    int[][] x = new int[n][n];

    obtineConfiguratiaInitiala(x);

    if (existaSolutie(x))
    {
      cautaSolutie(x);
    }
```

```
    else
    {
      System.out.println("NU exista solutii!!!");
    }
  }

  /** Citeste configuratia initiala a jocului. */
  public static void obtineConfiguratiaInitiala(int[][] x)
  {
    for (int i = 0; i < x.length; i++)
    {
      for (int j = 0; j < x.length; j++)
      {
        System.out.print("x[" + i +
          "][" + j + "]=");
        x[i][j] = Reader.readInt();
      }
    }
  }

  /** Determina daca jocul are solutie. */
  public static boolean existaSolutie(int[][] x)
  {
    int s = 0;
    for (int i = 1; i <= 16; i++)
    {
      s += less(x, i);
    }

    s += determinaPozSpatiu(x);

    if (s % 2 == 0) return true;
    else return false;
  }

  /** Functia position. */
  public static int position(int[][] x, int el)
  {
    for (int i = 0; i < x.length; i++)
    {
      for (int j = 0; j < x.length; j++)
      {
        if (el == x[i][j])
        {
          return i * x.length + j + 1;
        }
      }
    }

    return 0;
```

```
}

/** Functia less. */
public static int less(int[][] x, int i)
{
  int nr = 0;

  for (int j = 1; j < i; j++)
  {
    if (position(x, j) > position(x, i))
    {
      nr++;
    }
  }

  return nr;
}

/** Determina pozitia spatiului pe grila de joc. */
public static int determinaPozSpatiu(int[][] x)
{
  for (int i = 0; i < x.length; i++)
  {
    for (int j = 0; j < x.length; j++)
    {
      //4 * 4 = 16 = spatiu liber
      if (x[i][j] == x.length * x.length)
      {
        if ((i % 2 == 0 && j % 2 == 1) ||
            (i % 2 == 1 && j % 2 == 0))
        {
          return 1;
        }
        else
        {
          return 0;
        }
      }
    }
  }

  return 0;
}

/** Cauta solutia jocului pe baza configuratiei initiale. */
public static void cautaSolutie(int[][] x)
{
  //lista nodurilor active
  Vector noduriActive = new Vector();
  //nodul expandat (implicit este configuratia initiala)
```

```
    int[][] nodExpandat = x;

    //pastreaza succesiunea de mutari efectuate
    Vector mutari = new Vector();

    for (; ;)
    {
      distanta++;
      noduriActive = determinaNoduriActive(nodExpandat,
                     noduriActive);

      if (noduriActive.size() == 0) return;

      nodExpandat = determinaNodExpandat(noduriActive);

      mutari.addElement(nodExpandat);
      if (gc(nodExpandat) == 0)
      {
        afiseazaSolutie(mutari);
        return;
      }
      int minPos = noduriActive.indexOf(nodExpandat);
      noduriActive.removeElementAt(minPos);
      valoriF.removeElementAt(minPos);
    }
  }

  /**
   * Determina nodurile active din multimea carora se
   * va alege nodul expandat.
   */
  public static Vector determinaNoduriActive(int[][] x,
                 Vector noduriActive)
  {
    //cauta spatiul gol in interiorul configuratiei
    int i1 = 0;
    int j1 = 0;

    for (int i = 0; i < x.length; i++)
    {
      for (int j = 0; j < x.length; j++)
      {
        if (x[i][j] == x.length * x.length)
        {
          i1 = i;
          j1 = j;
        }
      }
    }

```

```
if (i1 != 0) //mutare SUS posibila
{
  int[][] xUp = new int[x.length][x.length];

  copie(x, xUp);

  int aux = xUp[i1 - 1][j1];
  xUp[i1 - 1][j1] = xUp[i1][j1];
  xUp[i1][j1] = aux;

  noduriActive.addElement(xUp);
  valoriF.addElement(new Integer(distanta));
}

if (i1 != x.length - 1) //mutare JOS posibila
{
  int[][] xDown = new int[x.length][x.length];

  copie(x, xDown);

  int aux = xDown[i1][j1];
  xDown[i1][j1] = xDown[i1 + 1][j1];
  xDown[i1 + 1][j1] = aux;

  noduriActive.addElement(xDown);
  valoriF.addElement(new Integer(distanta));
}

if (j1 != 0) //mutare STANGA posibila
{
  int[][] xLeft = new int[x.length][x.length];

  copie(x, xLeft);

  int aux = xLeft[i1][j1 - 1];
  xLeft[i1][j1 - 1] = xLeft[i1][j1];
  xLeft[i1][j1] = aux;

  noduriActive.addElement(xLeft);
  valoriF.addElement(new Integer(distanta));
}

if (j1 != x.length - 1) //mutare DREAPTA posibila
{
  int[][] xRight = new int[x.length][x.length];

  copie(x, xRight);

  int aux = xRight[i1][j1];
  xRight[i1][j1] = xRight[i1][j1 + 1];
```

```
    xRight[i1][j1 + 1] = aux;

    noduriActive.addElement(xRight);
    valoriF.addElement(new Integer(distanta));
  }
  return noduriActive;
}

/**
 * Metoda utila de copiere a valorilor dintr-un sir sursa
 * intr-un sir destinatie.
 */
public static void copie(int[][] src, int[][] dest)
{
  for (int i = 0; i < src.length; i++)
  {
    for (int j = 0; j < src.length; j++)
    {
      dest[i][j] = src[i][j];
    }
  }
}

/** Determina nodul expandat pe baza nodurilor active. */
public static int[][] determinaNodExpandat(Vector noduriActive)
{
  int minPos = 0;
  int min = Integer.MAX_VALUE;

  for (int i = 0; i < noduriActive.size(); i++)
  {
    int[][] el = (int[][]) noduriActive.elementAt(i);

    if (cc(el, i) < min)
    {
      min = cc(el, i);
      minPos = i;
    }
  }

  return (int[][]) noduriActive.elementAt(minPos);
}

/** Calculul functiei c. */
public static int cc(int[][] x, int j)
{
  return gc(x) + f(j);
}

/** Calculul functiei g. */
```

```
public static int gc(int[][] x)
{
  int nr = 0;

  for (int i = 0; i < x.length; i++)
  {
    for (int j = 0; j < x.length; j++)
    {
      if (x[i][j] != i * x.length + j + 1 &&
        x[i][j] != 16)
      {
        nr++;
      }
    }
  }

  return nr;
}

/** Calculul functiei f. */
public static int f(int j)
{
  return ((Integer) valoriF.elementAt(j)).intValue();
}

/** Afisarea solutiei obtinute. */
public static void afiseazaSolutie(Vector mutari)
{
  System.out.println("Mutarile efectuate: ");
  for (int i = 0; i < mutari.size(); i++)
  {
    int[][] m = (int[][]) mutari.elementAt(i);

    for (int j = 0; j < m.length; j++)
    {
      for (int k = 0; k < m.length; k++)
      {
        if (m[j][k] == 16) //spatiu
        {
          System.out.print("    ");
        }
        else if (m[j][k] < 10)
        {
          System.out.print("   " + m[j][k]);
        }
        else
        {
          System.out.print("  " + m[j][k]);
        }
      }
```

```
            System.out.println();
          }
          System.out.println();
        }
      }
    }
```

Rezumat

Capitolul de faţă a prezentat metoda *Branch & bound* de elaborare a algoritmilor. Asemănătoare metodei *Backtracking*, această metodă este mai rar utilizată decât Backtracking-ul, de aceea a fost prezentată în mai puţine detalii. S-a insistat însă asupra mecanismului ei de funcţionare, care se bazează pe o funcţie de cost asociată nodurilor din arborele de stări. Funcţia de cost trebuie aleasă în funcţie de problema care trebuie rezolvată. Problema puzzle-ului cu 15 elemente este un bun exemplu pentru aprofundarea cunoştinţelor de *Branch and bound*.

Noţiuni fundamentale

arbore de stări: arbore oarecare generat de aplicarea mutărilor permise de problemă unei configuraţii iniţiale.

funţie de cost: funcţie specială care diferă de la problemă la problemă şi este asociată unui nod din arborele de stări. Cu ajutorul ei se elimină subarborii care nu duc la rezultatul dorit, obţinându-se o reducere semnificativă a timpului de aflare a soluţiei.

nod activ: nod care a fost obţinut prin expandarea unui alt nod şi care nu a fost încă, la rândul lui, expandat.

nod expandat: nodul curent, pentru care au fost generaţi fiii (s-au realizat mutările permise de problemă)

nod răspuns: configuraţia finală la care trebuie să se ajungă prin efectuarea mutărilor permise asupra configuraţiei iniţiale.

Exerciţii

Teorie

1. Fie $S = x + \sum_{i=1}^{16} less(i)$ pentru o configuraţie oarecare a problemei puzzle cu 15 elemente. Demonstraţi că $S \mod 2$ este invariant pentru

orice stare obţinută din starea iniţială. Cu alte cuvinte, dacă S este impară, atunci S rămâne impară pentru orice secvenţă de mutări legală, iar dacă este pară, atunci rămâne pară.

2. Folosind observaţia de la exerciţiul precedent, demonstraţi teorema 15.2.1.

În practică

1. Rezolvaţi problema discretă a rucsacului folosind o abordare *Branch and bound*. Reamintim că în problema discretă a rucsacului se dau un număr de *n* produse, fiecare cu greutatea w_i şi costul p_i, şi un număr pozitiv *M* ce reprezintă capacitatea rucsacului. Se cere să se aleagă o submulţime de produse astfel încât $\sum_1^n w_i x_i \leq M$ şi $\sum_1^n p_i x_i$ este maxim, unde *x* este un vector de dimensiune *n* ale cărui elemente se află în intervalul [0, 1]. Altfel spus, este posibil ca dintr-un produs să se aleagă doar o anumită parte şi nu întregul.

2. Rezolvaţi problema comis-voiajorului folosind metoda *Branch and bound*. Reamintim că problema comis-voiajorului are următoarea definire: un comis voiajor trebuie să treacă prin toate oraşele dintr-un judeţ, doar o singură dată, astfel încât costul drumului realizat să fie minim.

3. Se dă următoarea problemă de optimizare:
   $\max 3x_1 + x_2 + x_3 + 2x_4 + 2x_5 + 3x_6$
   având restricţiile:
   $$\begin{array}{lllllll}
x_1 & +x_2 & & & & & +y_1 = 1 \\
x_1 & & +x_3 & & & & +y_2 = 1 \\
 & x_2 & +x_3 & & & & +y_3 = 1 \\
 & x_2 & & +x_4 & & & +y_4 = 1 \\
 & & x_3 & & +x_5 & & +y_5 = 1 \\
 & & & x_4 & +x_5 & & +y_6 = 1 \\
 & & & x_4 & & +x_6 & +y_7 = 1 \\
 & & & & x_5 & +x_6 & +y_8 = 1
\end{array}$$
   $x_i \in \{0, 1\}, y_i \geq 0\, i = 1, 2, \ldots, 6$
   Observaţi că spaţiul de căutare al problemei este finit (2^6). Să se găsească soluţia optimă a problemei utilizând mai întâi o strategie de tip Backtracking, iar apoi o strategie tip Branch & bound.

16. Metode de elaborare a algoritmilor (sinteză)

Punctul de convergenţă al artei şi ştiinţei este metoda.

Edward Bulwer-Lytton

Vom prezenta o scurtă sinteză a metodelor de elaborare a algoritmilor, care poate fi utilizată ca o referinţă rapidă în clasificarea problemelor de algoritmică.

16.1 Backtracking

Metoda backtracking se poate aplica problemelor a căror soluţie se scrie sub formă de şir, cu fiecare componentă a şirului aparţinând unei mulţimi finite. Soluţia se construieşte incremental, începând cu prima componentă şi mergând către ultima, cu eventuale reveniri la componentele anterioare. Fiecare componentă trebuie să respecte condiţiile de continuare, care sunt derivate din condiţiile interne.

Complexitatea algoritmilor backtracking este în general exponenţială. Cu cât condiţia de continuare este mai bine aleasă (elimină cât mai multe soluţii parţiale care nu pot conduce la o soluţie), cu atât soluţia va fi mai eficientă.

16.2 Divide et impera

Divide et impera se aplică problemelor care se pot descompune în subprobleme, astfel încât soluţia problemei originale să se poată obţine uşor din soluţiile subproblemelor. Subproblemele se împart la rândul lor în subprobleme până

când se ajunge la subprobleme triviale, care admit soluţie imediată. Exprimarea rezolvării este recursivă, ca şi algoritmii care utilizează această metodă.

Timpul de calcul este, în general, de forma $n^k \log n$. Cele mai eficiente metode de sortare (Mergesort, Quicksort) se bazează pe această metodă.

16.3 Greedy

Metoda Greedy se aplică problemelor de optimizare care respectă principiul optimalităţii (substructură optimă) şi principiul alegerii Greedy. Principiul optimalităţii ne asigură că o soluţie optimă cuprinde subsoluţii optime. Principiul alegerii Greedy ne asigură că alegând la fiecare pas optimul local, se obţine optimul global. Există multe situaţii în care metoda Greedy se aplică problemelor care nu respectă principiul alegerii Greedy (de exemplu, problema discretă a rucsacului), caz în care soluţia dată de metoda Greeddy nu va mai fi întotdeauna cea optimă.

Datorită simplităţii strategiei, soluţiile Greedy au cel mai adesea complexitate polinomială.

16.4 Programare dinamică

Programarea dinamică se aplică problemelor de optimizare care respectă principiul optimalităţii. Reiese imediat că programarea dinamică se aplică unei clase mai largi de probleme decât metoda Greedy. Problemele de programare dinamică se rezolvă prin construirea soluţiei problemei pe baza soluţiilor optime ale subproblemelor. Din acest punct de vedere, programarea dinamică se află între Greedy, care ia în considerare o singură subproblemă (cea optimă local), şi Backtracking, care generează exhaustiv toate soluţiile.

Complexitatea algoritmilor de programare dinamică este în general polinomială (vezi toate problemele prezentate în capitolul dedicat metodei Greedy), dar poate fi şi exponenţială (problema comis-voiajorului).

Abordarea rezolvării este bottom-up, ceea ce înseamnă că se rezolvă mai întâi problemele de dimensiune mai mică, pe baza lor cele mai mari, până când se ajunge la soluţia problemei iniţiale.

16.5 Branch & bound

Branch & bound este o variantă a metodei backtracking, în care alegerea următorului element nu se face pur şi simplu aleator (sau în ordine crescătoa-

re) ci se urmăreşte alegerea variantelor care sunt mai promiţătoare în vederea obţinerii unei soluţii. Aceasta înseamnă că fiecare posibilă variantă este evaluată după anumite criterii specifice fiecărei probleme în parte, alegându-se la fiecare pas varianta evaluată a fi optimă.

Tabel sinteză:
Tabelul următor prezintă o sinteză a informaţiilor din acest paragraf.

Metoda	Condiţii de aplicare	Timp de lucru (în general)	Construire soluţie
Backtracking	Şir cu elemente luând valori în mulţimi finite	exponenţial	incremental
Divide et Impera	Soluţia se poate construi pe baza soluţiilor subproblemelor	$n^k * logn$	top-down
Greedy	Probleme de optimizare care respectă principiul optimalităţii şi al alegerii Greedy	polinomial	incremental
Programare dinamică	Probleme de optimizare care respectă principiul optimalităţii	polinomial	bottom-up
Branch & bound	Probleme de optimizare dificile în care aplicarea celorlalte metode nu este adecvată.	diferă funcţie de problemă	parcurgere least-cost

Bibliografie

[Andonie] R. Andonie, I. Gârbacea - Algoritmi şi Calculabilitate, Computer Press Agora, 1995

[Balanescu] T.Bălănescu - Metoda Backtracking, Gazeta de informatică, nr 2/1993, pag. 9-18

[Boian] F.M. Boian - De la aritmetică la calculatoare, Presa Universitară Clujeană, 1996, ISBN 973-97535-5-8

[Cormen] T.H. Cormen, C.E. Leiserson, R.R. Rivest - Introducere în Algoritmi, Computer Press Agora, 2000, ISBN 973-97534-3-4

[Danciu] D.Danciu, S.Dumitrescu - Algoritmică şi Programare, Curs şi Probleme de Seminar, Reprografia Univ. Transilvania, 2002

[Eckel] B. Eckel - Thinking in Java, http://www.mindview.net/Books/TIJ/

[Horrowitz] E. Horrowitz, S. Sahni - Fundamentals of Computer Algorithms, Computer Science Press, ISBN 3-540-12035-1

[Norton] P. Norton, W. Stanek - Peter Norton's Guide to Java Programming, Sams.net Publishing, 1996, ISBN 1-57521-088-6

[Roman] Ed Roman - Mastering Enterprise JavaBeans, Wiley Computer Publishing, 1999, ISBN 0-471-33229-1

[Sorin] T.Sorin - Tehnici de programare, manual pentru clasa a X-a, L&S Infomat, 1996

[Tomescu] I. Tomescu - Data Structures, Bucharest University Press, Bucharest, 1997

[Vanderburg] G. Vanderburg - Tricks of the Java Programming Gurus, Sams.net Publishing, 1996, ISBN 1-57521-102-5

[Weiss] M.A. Weiss - Data Structures and Problem Solving Using Java, Addison-Wesley, 1998, ISBN 0-201-54991-3

Index